SPVU005

TMS34010 C Compiler User's Guide

TMS34010 C Compiler User's Guide

Graphics Products

TEXAS INSTRUMENTS

TMS34010 C Compiler User's Guide

TEXAS INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes in the devices or the device specifications identified in this publication without notice. TI advises its customers to obtain the latest version of device specifications to verify, before placing orders, that the information being relied upon by the customer is current.

In the absence of written agreement to the contrary, TI assumes no liability for TI applications assistance, customer's product design, or infringement of patents or copyrights of third parties by or arising from use of semiconductor devices described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor devices might be or are used.

Contents

Section *Page*

Illustrations

Tables

1. TMS34010 C Compiler Package Product Overview

The C programming language has become one of the dominant languages for high-level graphics software. Thus, C is a logical choice for systems, peripheral, and applications software developers using the TMS34010 Graphics System Processor. GSP C, a compiler implementation for the TMS34010, has been created to allow developers to tap this software base for high-performance TMS34010 systems. GSP C accepts standard Kernighan and Ritchie C source code and produces TMS34010 assembly language source code.

The TMS34010 C Compiler Package consists of a C Preprocessor, a Parser, and a Code Generator, as well as Run-Time Support and Floating Point Libraries. The TMS34010 Assembly Language Package supports the assembly, archiving, and linking of compiler-generated software. The resulting executable code can be run on a TMS34010 target system. Target systems can include software simulators, software development boards, Texas Instruments XDS (Extended Development Systems) units, or custom TMS34010 systems. An illustration of this interaction is shown in Figure 1-1.

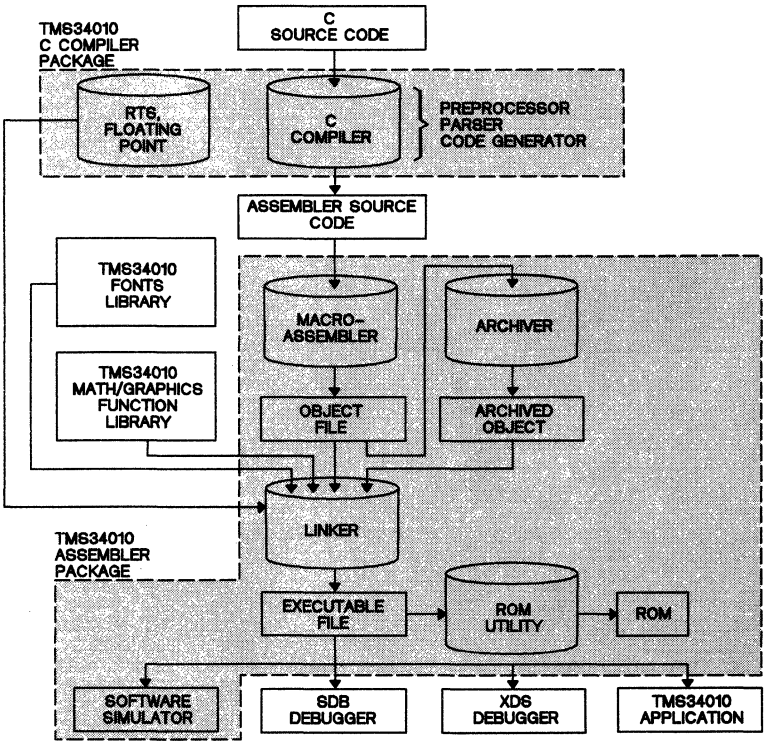

Figure 1-1. TMS34010 C Compiler Package Interaction

1.1 Applicable Documents

- Kernighan, B., and D. Ritchie. *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
- Kochan, Steve G. *Programming in C*, Hayden Book Company.
- Sobelman, Gerald E. and David E. Krekelberg. *Advanced C: Techniques and Applications*, Que Corporation, 1985.
- *TMS34010 User's Guide* (SPPU005), covering all the hardware related subjects: architecture, registers, addressing modes, etc.
- *TMS34010 Assembly Language Tools User's Guide* (SPDU076), covering the assembler, linker, archiver, and PROM programming utility.
- *TMS34010 Software Development Board User's Guide* (SPVU002), describing a PC-based tool for developing and debugging programs for the TMS34010.
- *TMS34010 XDS/22 User's Guide* (SPDU058) describing the most powerful tool available for developing software for the TMS34010.
- *TMS34010 Font Library User's Guide* (SPVU005), describing the font data structure and illustrating each of the available fonts.

1.2 Documentation Conventions

The documentation conventions used in this book include:

<> Angle brackets enclose user-supplied information that is to be typed out; for example, **<filename>** indicates that the name of the file is to be entered. The brackets themselves are not entered, except in the case of *#include* statements used in illustrating the syntax of some macros (see Section 6, TMS34010 Run-Time Support).

[] Square brackets enclose optional items.

A `special font` is used for information displayed on the screen.

Underscoring is used to indicate the information you type in response to prompts or other screen displays.

Italics are used to highlight function names, commands, code, and similar items within a text line.

Boldface type is used to indicate emphasis.

In Section 4, text in the left margin gives references to corresponding sections in Kernighan and Ritchie's book, *The C Programming Language*, or indicates additional information not included in their book.

2. TMS34010 C Compiler Package Installation

The TMS34010 C Compiler operates under these operating systems:

- MS[1]-DOS Version 2.1 (or later)
- PC-DOS[2] Version 2.1 (or later)
- VAX/VMS[3]
- VAX/ULTRIX™[3]
- VAX/UNIX™[4] System V

---

1 MS-DOS is a trademark of Microsoft Corporation.

2 PC-DOS is a trademark of International Business Machines.

3 VAX, VMS, and ULTRIX are trademarks of Digital Equipment Corporation.

4 UNIX is a trademark of AT&T Bell Laboratories.

2.1 Installation for Texas Instruments and IBM PC Systems

2.1.1 Hardware Requirements

The TMS34010 C Compiler requires your system to be configured with 512K bytes of RAM. It is also recommended that the system be configured with a Winchester disk.

> **Note:**
>
> The MS-DOS/PC-DOS version of the TMS34010 C compiler is distributed on two floppy diskettes. The first step for the user is to make backup copies of the software on blank diskettes. The backup diskettes should be used in the system instead of the original diskettes. This will afford the advantage of keeping the original copy free from accidental corruption.

2.1.2 Installation for TIPC and IBM PC Systems with Dual Diskette Drives

> **Warning:**
>
> **Prior to proceeding, make sure the distribution diskettes have write-protect tabs applied to prevent the risk of destroying the information on the diskettes.**

Use the DISKCOPY command to back up the software package onto your blank diskettes. Follow these steps to perform this backup:

1) Insert the MS-DOS or PC-DOS system diskette in drive A:.

2) Enter:

   DISKCOPY A: B:/F/V

3) The DISKCOPY function prompts you to place the source diskette in drive A:. Place one of the distribution diskettes in drive A:.

4) You are now prompted to place the destination diskette in drive B:. Place a blank diskette in drive B:.

   The DISKCOPY function copies from the distribution diskette in drive A: to the blank diskette in drive B:. The /F option formats the destination diskette before copying, precluding the possibility of directory errors on the backup diskette. The /V option checks the diskette for media errors.

5) Repeat steps 2 through 5 for each distribution diskette.

2.1.3 Installation for PC Systems with a Winchester Disk and Single Diskette Drive

This procedure is for backup of the distribution diskettes on a second set of floppy diskettes using a single floppy disk drive. Additionally, it is recommended that the distribution diskettes be backed up on the Winchester, both for speed of access, as well as for safety of the software.

MS-DOS 2.1 and PC-DOS 2.1 are operating system versions that support a hierarchical file system. For these systems it is recommended that a subdirectory for TMS34010 development tools be created.

> **Warning:**
>
> **Prior to proceeding, make sure the distribution diskettes have write-protect tabs applied to prevent the risk of destroying the information on the diskettes.**

To back up the distribution diskettes using the single floppy disk drive, follow these steps:

1) Making sure that the utility function DISKCOPY.COM is available for execution, type:

   DISKCOPY A: B:/F/V

2) The DISKCOPY function prompts you to place the source diskette in drive A:. Place one of the distribution diskettes in drive A:. The DISKCOPY function accesses the disk to determine its type and format.

3) You are then prompted to replace the source diskette in drive A: with the destination diskette. Place a blank diskette in drive A:. The /F option of the DISKCOPY function causes the destination diskette to be formatted before copying, precluding the possibility of directory errors on the backup diskette. The /V option checks the diskette for media errors.

4) After DISKCOPY formats the backup diskette, it prompts you to switch back and forth between the source and destination diskettes, copying the distribution diskette to the backup diskette. After each distribution diskette has been copied, DISKCOPY prompts you for more diskettes to copy. Respond YES until all the distribution diskettes are copied.

If you plan to place the TMS34010 software tools in a directory on the Winchester disk as recommended, follow these steps:

1) Create a subdirectory to contain the TMS34010 development tools by typing:

   MKDIR \GSPTOOLS

2) Make the TMS34010 development tools directory the current directory by typing:

   CD \GSPTOOLS

3) Place a distribution diskette in drive A:. Copy the contents of the distribution diskette by typing:

COPY A:*.* /V

The /V option causes the system to verify that, after copying, the source and the destination files are identical.

4) Repeat step 3 for each distribution diskette.

5) Make sure the directory you placed the tools in (in this example, GSPTOOLS) is in the current pathway.

2.1.4 Tools Verification on PC Systems

When the TMS34010 development tools have been backed up and/or copied onto the Winchester disk, verify that your copies are executable. Use the batch file GSPC.BAT provided in the package for this verification. This batch file first calls the C preprocessor (GSPCPP), then the parser (GSPCC), the code generator next (GSPCG), and finally the assembler (GSPA). The result is linkable object code. When using this batch file, remember that you must have installed the TMS34010 Assembly Language Package on your system; this package includes the assembler (GSPA), the linker (GSPLNK), the simulator (GSPSIM), and the archiver (GSPAR).

1) To verify that the compiler is properly installed, invoke GSPC.BAT to compile the file TESTC34.C by typing:

GSPC TESTC34

2) The batch file responds with:

```
---[TESTC34]---
C Pre-Processor,      <version number>
© Copyright  1985, 1986  Texas Instruments Inc.

GSP C Compiler,        <version number>
© Copyright  1985, 1986  Texas Instruments Inc.
"TESTC34.C": ==> MAIN

GSP C Codegen,         <version number>
© Copyright  1985, 1986  Texas Instruments Inc.
"TESTC34.C": ==> MAIN

GSP COFF Assembler,  <version number>
© Copyright  1985, 1986  Texas Instruments Inc.
   PASS 1
   PASS 1.1 ON SECTION  .text
   PASS 2

   No Errors, No Warnings
```

3) Verify the operation of the TMS34010 linker. The link command file TESTC34.CMD is included in the package for linking the output of the compiler and assembler together with the runtime support (RTS.LIB) and floating point (FLIB.LIB) libraries. Type:

GSPLNK TESTC34.CMD

The TMS34010 linker responds with:

```
GSP COFF Linker, <version number>
© Copyright 1985, 1986  Texas Instruments, Inc.
```

where <version number> is the version number of the current TMS34010 linker.

4) Continue verification of the TMS34010 development tools by invoking the TMS34010 simulator. Type:

GSPSIM TESTC34

The TMS34010 simulator responds by loading the linked output TESTC34.OUT and displaying the register file and status information. After the simulator's display has been placed on the screen, run the linked output by typing:

RUN <return>

The simulator responds by displaying:

```
YOU'RE OK, WE'RE OK!!!
```

in white letters in the center of the scratch display area. Exit the simulator by typing:

Q* <return>

5) Conclude the verification session by testing the object/source archiver. Type:

GSPAR t RTS.LIB

The archiver responds by giving a table of contents for the runtime support library, similar to the following:

```
GSP Archiver          <version number>
© Copyright 1985, 1986  Texas Instruments, Inc.

FILE NAME        SIZE      DATE
  atof.obj       1216      Tue Aug 12 00:01:04 1986
  atoi.obj        450      Tue Aug 12 00:01:04 1986
  boot.obj        572      Tue Aug 12 00:01:04 1986
 ctype.obj        464      Tue Aug 12 00:01:04 1986
  ltoa.obj        424      Tue Aug 12 00:01:04 1986
memory.obj       1874      Tue Aug 12 00:01:04 1986
movmem.obj        382      Tue Aug 12 00:01:04 1986
printf.obj        878      Tue Aug 12 00:01:04 1986
strcat.obj        358      Tue Aug 12 00:01:04 1986
strchr.obj        352      Tue Aug 12 00:01:04 1986
strcmp.obj        360      Tue Aug 12 00:01:04 1986
strcpy.obj        350      Tue Aug 12 00:01:04 1986
strlen.obj        340      Tue Aug 12 00:01:04 1986
strncat.obj       374      Tue Aug 12 00:01:04 1986
strncmp.obj       376      Tue Aug 12 00:01:04 1986
strncpy.obj       372      Tue Aug 12 00:01:04 1986
strrchr.obj       352      Tue Aug 12 00:01:04 1986
fconvert.obj      342      Tue Aug 12 00:01:04 1986
 setjmp.obj       332      Tue Aug 12 00:01:04 1986
```

2.2 VAX/VMS Systems

The tape provided was made at 1600 BPI using the VMS BACKUP utility. Follow the instructions in this section to install and verify the tools contained on the tape.

2.2.1 Installation Procedure

1) Mount the tape on your tape drive.

2) Execute the following VMS commands. Note that you must create a destination directory in which to put the tools. This example uses DEST:<dest-directory> to indicate the destination. TAPE: is to be replaced with the name of the tape drive being used in the backup procedure.

```
$ allocate        TAPE:
$ init/den=1600   TAPE:gspc
$ mount/for       TAPE:
$ backup          TAPE:gspc.bck   DEST:<dest-directory>
$ dismount        TAPE:
$ dealloc         TAPE:
```

3) The product tape contains a command file called *setup.com*. This file is used to set up VMS symbols that allow the tools to be executed in the same manner as any other command. This command file must be run as follows:

```
$ @setup DEST:<dest-directory>
```

This sets up symbols to be used to call the various tools. The symbols defined are listed on the screen by the command file.

2.2.2 Tools Verification

The product tape contains a test file (*testc34.c*) that can be compiled and compared with *testc34.cmp* to ensure that the C compiler has been installed correctly and is functioning.

For example, enter:

```
$ gspcpp testc34
$ gspcc  testc34
$ gspcg  -* testc34
```

This produces an assembly language file *testc34.asm*, which should be exactly like the file *testc34.cmp* shipped on the product tape.

2.3 VAX/ULTRIX and VAX/UNIX System V

The tape provided was made at 1600 BPI using the *tar* utility. This section instructs you in installing and verifying the tools contained on the tape.

2.3.1 Installation Procedure

1) Mount the tape on your tape drive.

2) Make the current directory the directory into which the tools will be restored.

3) Enter the *tar* command appropriate for your system; for example:

   tar vx

   This copies the entire tape into the current directory.

2.3.2 Tools Verification on ULTRIX and UNIX Systems

The product tape contains a test file (*testc34.c*) that can be compiled to ensure that the working C compiler has been installed. The tape contains the correct output file to compare to the file created during the test.

For example, enter:

```
gspcpp testc34
gspcc testc34
gspcg -\* testc34
```

This creates the assembly file *testc34.asm*, which can be compared to the *testc34.cmp* file shipped on the tape.

3. Invocation and Operation of the TMS34010 C Compiler

The TMS34010 C compiler consists of three parts:

- Preprocessor (GSPCPP)
- Parser (GSPCC)
- Code generator (GSPCG)

The output of the code generator must be assembled with the TMS34010 COFF (common object file format) assembler and linked with TMS34010 C Run-Time Support using the COFF linker (GSPLNK) to produce executable object code. Refer to the *TMS34010 User's Guide* and UNIX COFF documentation for details of the object file format.

> **Note:**
>
> Each of the tools discussed here has options available for use in the invocation (command) line. These options are unique among themselves and must be used as shown herein. Any other items included in the invocation may be ignored or may cause errors.

3.1 The TMS34010 C Preprocessor (GSPCPP)

The C preprocessor, GSPCPP, is invoked as the first pass of the TMS34010 C compiler. GSPCPP handles macro definitions and substitutions, *#include* files, line number directives, and conditional compilation. The preprocessor requires the original C source file as input and produces as output a modified source file suitable for input to GSPCC.

This preprocessor is the same preprocessor described in *The C Programming Language*, by B. Kernighan and D. Ritchie (referred to hereafter as K&R). Additional information can be found in that book.

3.1.1 Invoking GSPCPP

The preprocessor begins execution when the following command is entered:

```
gspcpp [input file] [output file] [options]
```

where:

input file is the name of the C source file used as the input file. If no extension is given, an extension of *.c* is assumed. If no input file is given, a prompt appears.

output file is the name of the file output by GSPCPP. The *output file* can be omitted, in which case the name given to the output-file defaults to the input-file name with an extension of *.cpp*.

options lists any of the available GSPCPP options (see Section 3.1.3). The *options* parameter can appear anywhere in the invocation line and can be used multiple times.

3.1.2 GSPCPP Options

GSPCPP's options are **case-sensitive**, single-letter fields prefixed by hyphens (-). Some options have additional fields, which immediately follow the option letter with **no intervening spaces**. The following list describes options available; each option is uppercase.

-P Tells the preprocessor not to produce the line number and file :i3-P information used by the compiler. The compilation is otherwise the same as if this information were produced.

-C Copies comments to the output file. Otherwise, GSPCPP strips comments.

-D<name>=def Defines *name* as if the following line appeared at the top of the input file:

```
#define name def
```

The symbol *name* is then recognized for *#if* and *#ifdef* statements, without having been explicitly defined in the text of the program.

The =*def* can be omitted, in which case *name* is defined to the value 1.

This option can be used multiple times for defining multiple names by separating the *-D* options by spaces.

-I<dir> (Uppercase *i*) Adds *dir* to the list of directories to be searched for *#include* files. Omitting *dir* causes option *-I* to be ignored. This option can be used multiple times, each one separated by a space, to search more directories.

3.1.3 Operation of GSPCPP

GSPCPP maintains and recognizes the name _ _LINE_ _ as the current line number (a decimal integer) and the name _ _FILE_ _ as the current file name (a C string). Note that these names begin with two underscore characters and end with two underscore characters **with no spaces between the characters**. You can use these names in the same way as any other defined name, including using them in macros.

All GSPCPP directives begin with the character #, which must appear in column 1. Any number of blanks and tabs are allowed between # and the directive name. The directives are described in Appendix C, and additional information can be found in K&R.

The error messages produced by the preprocessor are self-explanatory; the line number and the filename where the error occurred are printed along with the diagnostic.

3.2 The TMS34010 Parser (GSPCC)

The parser, GSPCC, is the second pass of the C compiler. It reads the output of GSPCPP (the preprocessor), parses it, performs syntax checking, and writes an intermediate file for input to the code generator (GSPCG).

3.2.1 Invoking the Parser

The parser is invoked with the command:

```
gspcc [input file] [output file] [-z]
```

where:

input file is the name of the input file. The default extension for the input file is *.cpp* if none is specified. If no input file is given, a prompt appears.

output file is the name of the output file. This is the intermediate file used as input to the GSPCG. If this argument is omitted, the intermediate file is given the same name as the input file with the default extension *.if*.

-z is the option which retains the *.cpp* input file. The *.cpp* file **is deleted** if *-z* is not specified. This parameter can be placed anywhere in the invocation line.

3.2.2 Operation of the Parser

The parser reads the C program (optionally preprocessed) and checks for errors, then outputs an intermediate file which is used as input to the code generator.

Most errors are fatal; that is, they prevent generation of an intermediate file and must be corrected before compilation can be completed. Some errors, however, merely produce warnings which hint of problems but do not prevent compilation.

As function definitions are encountered, the parser prints a progress message containing the name of the source file and the name of the function. For example:

```
"source.c": => main
```

The message lets you know how far along the compiler is in its execution and helps identify the location of errors.

After it finishes parsing, GSPCC deletes the input file if the input file's extension is *.cpp* and the *-z* option is not specified. Otherwise, the input file is kept.

3.3 The Code Generator (GSPCG)

The code generator (GSPCG) converts the intermediate code generated by the parser into assembly language source code suitable for direct input to the assembler or for modification with a text editor. The code produced by GSPCG is reentrant, relocatable, and can be stored in ROM.

3.3.1 Invoking GSPCG

The following command invokes GSPCG:

```
gspcg [input file] [output file] [tempfile][options]
```

where:

input file is the name of the input file, which is the file output by GSPCC with the extension *.if*. If this argument is not specified, a prompt appears.

output file is the name of the output file to be generated. The default name for the file is the name of the input file appended with the extension *.asm*.

tempfile is the name of the temporary file generated and used by the GSPCG. The default name is the input filename appended with extension *.tmp*. This file is deleted after use by GSPCG.

options lists any of the available GSPCG options (see Section 3.3.2, GSPCG Options). These options can appear anywhere in the invocation line and are not restricted to follow the list of filenames.

3.3.2 GSPCG Options

The options parameter in the invocation command is a character preceded by a hyphen in the format:

-x[x]...

where x represents an option character. These option characters can appear anywhere in the invocation line; they are not restricted to follow the list of filenames given. Options can be strung together in any order.

The options available are:

-a Indicates there may be assignments of the form **ptr=...* where *ptr* is a pointer to a named variable. A module containing assignments of this form can be compiled without this option, provided *ptr* does not point to a named variable. For example, if *ptr* points to an element of a dynamically- or statically-allocated array, the assignment of values to the array elements using the form **ptr=...* does not require compilation of the module with the *-a* option. Structures are also not considered to be named variables.

The compiler normally remembers that a register contains a constant or the value of a named variable, so it does not regenerate code to load that value into a register. The compiler also assumes that an assignment of the form **ptr=...* does not assign a value to a named variable. Because it cannot know the named variable that this assignment affects, if any, it must forget the contents of all registers it assumed contained values of named variables upon encountering such an assignment. Thus, when the *-a* option is used, the compiler generates less efficient code, because it forgets these registers' contents and has to regenerate the code. However, this is very rarely a problem.

-o Directs GSPCG to produce high-level-language debug directives in the output code. See the assembler section of the *TMS34010 Assembly Language Tools User's Guide* and the UNIX COFF documentation for more information about the types of debug information produced.

-v Directs GSPCG to produce code which can run in a multiprocess environment where all variables may be considered volatile. This flag should be used to compile modules which access variables that could be modified by another task (process). In general, code generated this way is somewhat less efficient.

-z Causes GSPCG to not delete the input file (intermediate file generated by GSPCC). This could be useful for generation of several object modules with different GSPCG options.

-r Causes GSPCG to periodically write a register status table to the output (assembly language) source file. The table is in the form of assembly language comments, and lists each register currently used by GSPCG. It also shows the type of each register's current contents. The table is printed between statements whenever the contents of registers could change. This is very useful if you want to modify the assembly language output.

An example:

```
*******************************************
* A5 - EXPRESSION FREE                    *
* A7 - USER VARIABLE FREE  SYMBOL = _ABC  *
* A9 - USER REGISTER USED                 *
*******************************************
```

* Instructs GSPCG to generate code which runs on certain preproduction units of the TMS34010. This option is not needed when compiling code for production units of the TMS34010. Certain instructions not available in preproduction units are not used when this option is invoked.

3.3.3 Input Requirements

GSPCG input must be the intermediate file produced by the parser. The output of the parser is fed, without modification, directly to the GSPCG.

3.3.4 GSPCG Output

GSPCG converts the intermediate file generated by the parser into assembly language source code suitable for input to the assembler or for modification with a text editor. This code is reentrant, relocatable, and can be stored in ROM.

3.4 Batch Execution of the C Compiler

The C compiler package contains the batch file GSPC.BAT, which executes the three phases of the compiler and the assembler. This batch file is invoked as follows:

```
gspc <input file>
```

where:

input file is the name of the C source file. The default extension of this file is *.c*. A prompt appears if no file name is given.

Example:

```
gspc PROGRAM
```

This example uses PROGRAM.C as the C source file and generates PROGRAM.ASM as the assembly file and PROGRAM.OBJ as the object file. PROGRAM.OBJ can be used as input to the linker, GSPLNK.

3.5 Assembling a C Program

The GSPC batch file automatically produces and assembles TMS34010 assembly language source from C programs. The assembly language source file is available under the name <input file>.ASM. If you wish to see an assembler listing of the file, you must explicitly assemble this file using the -*l* option of the GSP assembler.

Appendix E is an example showing the assembly language produced from C programs.

3.6 Archiving a C Program

C program object files may be archived using the GSPAR archiver program. Libraries should be organized so that all references to external symbols or functions are defined within the same library or in a following library. See the *TMS34010 Assembly Language Tools User's Guide* for more information.

3.7 Linking a C Program

Modular code is an important concept in writing software because it simplifies the tasks of debugging and porting. To make this modularization possible, the programmer must have the capability to link separate modules into one executable program. The TMS34010 C environment offers this capability by providing an assembler that produces object code which is linkable by the TMS34010 linker.

In the simplest case, a C program consisting of modules *prog1*, *prog2*, etc. can be linked to produce an executable output file called *prog.out* by invoking the linker as follows:

```
gsplnk -c -o prog.out prog1.obj prog2.obj ... rts.lib [flib.lib]
```

For further information, refer to the *TMS34010 Assembly Language Tools User's Guide.*

3.7.1 Run-Time Initialization

All C programs must be linked with an object module called *boot.obj*, which contains code and data for initializing the run-time environment. This is the first code executed when the program begins running, and it has the following responsibilities:

- Sets up the system stack.
- Processes the run-time initialization table and auto-initializes global variables.
- Disables interrupts and calls *_main*.

Boot.obj is supplied in the run-time support object library RTS.LIB. If you use the C code option with GSPLNK and include RTS.LIB in your link control file, *boot.obj* is automatically linked in (see Section 3.7.3, The -c Option in the Linker). Alternatively, you can use the archiver GSPAR to extract *boot.obj* from the library and link it in explicitly.

3.7.2 Object Libraries and Run-Time Support

The archive library RTS.LIB contains code for *boot.obj*, plus all additional C run-time support functions. These functions are described in Section 6, TMS34010 Run-Time Support. If your program uses any of these functions, RTS.LIB must be linked in with your object files.

Any program that uses floating point math must include the TMS34010 floating point library, which contains functions called by the compiled program to perform floating point operations. This library is called FLIB.LIB (see Appendix D for more information about the floating point library).

> **Note:**
>
> Programs that do not use floating point need not be linked with FLIB.LIB.

You can create your own object libraries and link them in. The linker operates so that only those modules from the library that define unresolved references will be included and linked. Refer to the detailed linker documentation in *TMS34010 Assembly Language Tools User's Guide*.

3.7.3 The -c Option in the Linker

The linker has an option *-c* to ease linking C programs. Use the *-c* option on the command line or in the link control file. The *-c* option has the following effects:

- Forces *_c_int00* to be the entry point. *_c_int00* is the start of the run-time initialization code in *boot.obj*. This also causes *boot.obj* to be included from the archive library since *boot.obj* is the module that defines the symbol *_c_int00*.
- Adds two bytes of zero padding at the end of the *.data* section. This is required to terminate the initialization tables.

3.7.4 Linker Command File

The following is an example of a typical command file for linking a C program. The C program consists of two C modules, *main.obj* and *sub.obj*, and an assembly language module, *asm.obj*. The program uses floating point and several routines from an archive library called *matrix.lib*.

```
-c                /* linking a C program  */
-m example.map    /* create a map file  */
-o example.out    /* specify name of output file */

main.obj          /* user's first C module */
sub.obj           /* user's second C module */
asm.obj           /* user's asm module */
flib.lib          /* floating point library */
rts.lib           /* run-time support library */
matrix.lib        /* user's archive library */
```

Example Description.

First, the desired linker options are listed:

-c Option used for linking C programs, as described in the previous section. If *-c* is not used, its effects **must be provided** using other linker directives.

-m Option used to create a map file. Follow the option with a string that represents a valid file name.

-o If no errors occur during link time, the linker creates an *.out* (executable) file. The default file name for this file is *a.out*. If you wish to alter this default file name, use the *-o* option followed by a string representing a valid file name.

Next, all the object files to be linked are listed, in any order. One of these files must define the symbol *main*, because *boot.obj* calls *main* as the start of your C program.

> **Note:**
>
> If any of the files are in assembly language, they may not contain *.data* sections because this will corrupt the C auto-initialization environment.

Any included *.obj* files are linked into the resultant *.out* file, whether they are used or not, while only referenced files from libraries are linked in.

Finally, all the archive object libraries that are to be searched are listed. Only modules from the libraries that resolve unresolved references are included in the output file.

4. The TMS34010 C Language

The C language compiled by the TMS34010 C compiler is based on the UNIX System V C language as described by K&R with some additions and clarifications. The most significant differences are:

- Addition of *enum* data type.
- Unique member names in structures not required.
- Pointers to fields within structures are allowed.
- Structures and unions may be passed as parameters to functions, returned by functions, and assigned directly.

This section is a comparison of the C language compiled by TMS34010 C and the C language described by Kernighan and Ritchie (K&R C).

> **Note:**
>
> Only the differences in the two forms of the C language are discussed here. The standard K&R C is followed except as contradicted by this section. Section numbers from K&R's "C Reference Manual", Appendix A of *The C Programming Language*, are shown in the left margin for reference.

4.1 Identifiers and Keywords

K&R 2.2 TMS34010 C treats the first 31 characters of an identifier as significant, as compared with eight in K&R C. This also applies to external names. **Case is significant:** uppercase characters are different from lowercase characters for identifier names in all TMS34010 C tools.

K&R 2.3 Three keywords exist in addition to the list in K&R. The new keywords are: *asm*, *void*, and *enum*.

4.2 Constants

K&R 2.4.1 All integer constants are of type int (signed, 32-bit length). Invalid digits in constants (i.e., 8 and 9 in octal) cause a warning message.

K&R 2.4.3 The escape code \v is recognized in character and string constants as a vertical tab character (ASCII code 11), in addition to the escape codes listed in K&R.

Added type Enumeration constants are a special type of integer constant not described by K&R. An identifier declared as an enumerator can be used wherever an integer constant can be used. See Section 4.7.4, page 4-6, for more information.

K&R 2.5 The maximum length of any string constant is 255 bytes in TMS34010 C, where the length is unlimited in K&R C.

All characters after an embedded null byte in a string constant are ignored; in other words, the first null byte terminates the string. However, this does not apply to strings used to initialize arrays of characters.

Identical string constants are stored as a single string, not as separate strings as in K&R C. However, this does not apply to strings used for any auto-initialization of arrays of characters.

4.3 TMS34010 C Data Types

K&R 4.0 The *char* data type is signed. A separate type, *unsigned char*, is supported. *Long* and *int* are functionally equivalent types, and either can be declared *unsigned*. The properties of *enum* types are identical to those of *unsigned int*. There is an additional type called *void*, which is used to declare a function that returns no value. The compiler checks that functions declared as *void* do not return values and that they are not used in expressions. Functions are the only type of objects that may be declared *void*.

4.3.1 Derived Types

TMS34010 C allows any type declaration to have up to six derived types. Constructions such as "pointer to", "array of", and "function returning" can be combined and applied only this number of times.

For example:

```
int (*(*n[][])())();
```

translates as "an array of arrays of pointers to functions returning pointers to functions returning integers." It has six derived types, the maximum allowed.

Structures, unions, and enumerations are not considered derived types for the purposes of these limits.

Also, the derived type cannot contain more than three array derivations. Note that each dimension in a multi-dimensional array is a separate array derivation; thus, arrays are limited to three dimensions in any type definition. However, types can be combined to produce any dimensioned array.

For example, the following construction declares *x* as a four-dimensional array:

```
typedef int dim2[][];
dim2 x[][];
```

K&R 2.6

Table 4-1 summarizes TMS34010 C data types:

Table 4-1. TMS34010 C Data Sizes

char	8 bits, signed ASCII	
unsigned char	8 bits, ASCII	
short	16 bits	
unsigned short	16 bits	
int	32 bits	
unsigned int	32 bits	
long	32 bits	
unsigned long	32 bits	
pointers	32 bits	
float	32 bits	range: $+-5.88 \times 10^{(-39)}$ through $+-1.70 \times 10^{38}$
double	64 bits	range: $+-1.11 \times 10^{(-308)}$ through $+-8.99 \times 10^{308}$
enum	1–32 bits - see Section 4.7.4	

4.4 Object Alignment

All objects except structure members and array members are aligned on a 16-bit (one word) boundary. In other words, with the exception of structure and array members, all objects begin at bit addresses whose four LSBs are zeros. In addition, because of the TMS34010's bit addressability, pointers can point to any bit address. Signed objects of less than 16 bits are sign-extended to 16 bits. Unsigned objects of less than 16 bits are zero-extended to 16 bits.

Structure or array members are not aligned to 16-bit boundaries. However, the structure or array itself begins at a 16-bit boundary. In the case of an array of structures, only the first structure in the array is constrained to begin on a 16-bit boundary.

For additional information on array alignment, see Structure Packing and Field Manipulation, 5-3.

4.5 Conversions

K&R 6.1

Integer objects are always widened to 32 bits when passed as arguments to a function. Signed objects of less than 32 bits are sign-extended to 32 bits; unsigned objects of less than 32 bits are zero-extended to 32 bits.

The type *char* is signed and is therefore sign-extended when widened to *integer* type. Sign extension can be disabled by using the type *unsigned char*.

K&R 6.3

Float and double types are converted to integer types by truncation.

All float variables are converted to doubles before arithmetic operations or before being passed as arguments to a function.

K&R 14.4

Pointers and integers (or longs) may be freely converted, since each occupies 32 bits of storage. Pointers to one data type can also be converted to pointers to another data type, since the TMS34010 has no alignment restrictions and all pointers are the same size.

4.6 Expressions

4.6.1 Void Expressions

Added type

A function of type *void* has no value (returns no value) and cannot be called in any way except as a separate statement or as the left operand of the comma operator. Functions can be declared or typecast as *void*.

4.6.2 Primary Expressions

K&R 7.1

In TMS34010 C, functions can return structures or unions. However, it is illegal to combine a function call with the structure-reference operator ".". Thus, primary expressions of the form *f(E).s* are not allowed. Note that this restriction does not apply to the indirect structure-reference operator "→", so that *f(E)→s* is legal.

The restriction of three array dimensions does not apply to expressions, because [ ] is treated as an operator.

4.6.3 Unary Operators in Expressions

K&R 7.2

The value yielded by the *sizeof* operator is calculated as the total number of bits used to store the object divided by eight. (Eight is the number of bits in a character.) *Sizeof* can be legally applied to *enum* objects and bit fields: if the result is not an integer, it is rounded up to the nearest integer.

4.6.4 Assignment Operators in Expressions

K&R 7.14

The obsolete C assignment operator *=op* is recognized by TMS34010 C, along with the current *op=*. Its use, however, causes a warning message to be issued.

4.7 Declarations

4.7.1 Storage Class Specifiers in Declarations

K&R 8.1

The first four local objects declared as *register* in a function will be stored in TMS34010 registers. A register can store *float* objects as well as objects of any integer or pointer type, and will significantly improve the efficiency of accessing the object. See also Section 5.2, Register Conventions, page 5-4.

Register variables declared as *short* or *char* are treated as *int*.

A *register* declaration of an invalid type or a declaration after the first four registers have been declared is treated as a normal *auto* declaration.

Note that functions' arguments declared as *register* are not implemented in this release. The register class is ignored for argument variables.

4.7.2 Type Specifiers in Declarations

K&R 8.2 In addition to the type-specifiers listed in K&R, objects may be declared with enum-specifiers. Enumerations are described in Section 4.7.4, page 4-6.

More type name combinations are allowed in TMS34010 C than in K&R C. The adjectives *long* and *short* may be used with or without the word *int*; the meaning is the same in either case. The word *unsigned* can be used in conjunction with any *integer* type or alone; if alone, *int* is implied. *Long float* is a synonym for *double*. Otherwise, only one type specifier is allowed in a declaration.

K&R 8.4 Contrary to K&R, functions may return structures or unions in TMS34010 C. In addition, structures and unions may be used as parameters to functions and may be directly assigned.

K&R 10 Formal parameters to a function may be declared as *struct* or *enum* (in addition to the normal function declarations), since TMS34010 C allows these types of objects to be passed to functions.

4.7.3 Structure and Union Declarations

K&R 8.5 Since the TMS34010 is bit-addressable, no alignment of any kind occurs for structure members. Ignore any comment in K&R about alignment or boundaries for structure members. Note that a field with width zero, normally used to force alignment, is ignored. It is true, however, that bit fields are limited to a width of 32 bits (see Object Alignment, page 4-4, and Structure Packing and Field Manipulation, page 5-3).

Any integer type may be declared as a field. Fields are treated as signed unless declared otherwise. Also, contrary to K&R, pointers to fields are legal in TMS34010 C.

K&R states that structure and union member names must be mutually distinct. In TMS34010 C, members of different structures or unions may have the same name. However, this requires that references to the member be fully qualified through all levels of nesting.

K&R 14.1 Again, TMS34010 C *allows assignment to and from structures, passing structures as parameters, and returning structures from functions.*

There is a comment in K&R regarding the compiler determining the type of a structure reference by the member name. Since member names are not required to be unique in TMS34010 C, this is not valid. All structure references must be fully qualified as members of the structure or union in which they were declared.

4.7.4 Enumeration Declarations

Additional

Enumerations allow the use of named integer constants in TMS34010 C. The syntax of an enumeration declaration is very similar to that of a structure or union. The keyword *enum* is substituted for *struct* or *union*, and a list of enumerators is substituted for the list of members.

Enumeration declarations have a "tag", as do structure and union declarations. This tag may be used in future declarations, without repeating the entire declaration.

The list of enumerators is simply a comma-separated list of identifiers. Each identifier may be either alone or followed by an equal sign and an integer constant. If no enumerators with "=" appear, then the values of the successive enumerators begin at zero and increase by one for each identifier. An identifier with an assigned value assumes that value, and subsequent enumerators continue counting by one from there. The assigned value may be negative, but counting still continues by positive one.

An object of type *enum* has a size determined as follows: if any of the object's enumerators have negative values, the object occupies 32 bits. Otherwise, the object occupies the minimum number of bits required to represent the largest enumerator value and is considered to be unsigned.

Unlike structure and union members, enumerators share their name space with ordinary variables and, therefore, must not conflict with variables or other enumerators in the same scope.

Enumerators may appear wherever integer constants are required and, therefore, can participate in arithmetic expressions, case expressions, and so forth. In addition, explicit integer expressions may be assigned to variables of type *enum*. The compiler does no range checking to insure the value will fit in the enumeration field. The compiler does, however, issue a warning message if an enumerator of one type is assigned to a variable of another.

Example:

```
enum color {red,blue,green=10,orange,purple=-2,cyan} x;
```

This statement declares x as a variable of type *enum* The enumerators (with their values in parentheses) are: *red* (0), *blue* (1), *green* (10), *orange* (11), *purple* (-2), *cyan* (-1). The variable *x* is allocated 32 bits because of the negative values.

All the following are legal operations:

```
x = blue;
x = blue + red;
x = 100;
i = red;  /*assume i has been declared an int*/
x = i + cyan;
```

4.8 Initialization of Static and Global Variables

K&R 8.6

An important difference between K&R C and TMS34010 C is that external and static variables are not pre-initialized to zero unless the program explicitly does so or it is specified as an option in the linker, GSPLNK.

If a program requires that external and static variables be pre-initialized, the linker can be used to do this. In the linker control file, use a fill value of 0 in the *.bss* section:

```
.
.
.
SECTIONS {
                .
                .
                .
                .bss{} = 0x00;
                }
.
.
.
```

4.9 *asm* Statement

Additional

TMS34010 C has another "statement" not mentioned in K&R: the *asm* statement. This statement allows assembly language text to be imbedded in the output assembler source file. The form of the *asm* statement is:

```
asm("<assembler text>");
```

The assembler text is enclosed in double quotes. All the usual character string escape codes have their normal meaning. The assembler text is copied directly to the assembler source file. Note that for GSPA, an assembler text statement without a label must begin with a blank.

The *asm* statement injects one line of assembly language into the output of the codegen. A series of *asm* commands places the instructions sequentially into the codegen output with no intervening code.

Asm statements do not follow the syntactic restrictions of normal statements and can appear anywhere in the C source, even outside blocks. However, they are ignored when they appear in a list of declarations.

Warning:

Extreme care must be taken not to disrupt the C environment with *asm* commands. No checking of the inserted instructions is done. Insertion of jumps and labels into C code may cause unpredictable results in variables manipulated in or around the inserted code. This command is provided so you can access features of the hardware, which by definition C is unable to access. Specifically, this command should not be used to change the value of a C variable; however, it can be used safely to read the current value of a variable.

In addition, the *asm* construct should not be used to insert assembler directives which would change the assembly environment.

The *asm* command is very useful in the context of register variables. A register variable is a variable in a C program that is declared by the user to reside in a machine register. The TMS34010 C compiler allows up to four machine registers to be allocated to register variables. These four registers, combined with the *asm* command, provide a means of manipulating data independently of the C environment.

4.10 Lexical Scope Rules

The lexical scope rules stated in K&R apply to TMS34010 C also, except that structures and unions each have distinct name spaces for their members. In addition, the name space of both enumeration variables and enumeration constants is the same as for ordinary variables.

5. TMS34010 C Run-Time Environment

5.1 Memory Model

TMS34010 C looks at memory as a single linear block partitioned into sub-blocks containing various components of data and program code. Each variable that is *external* or *static* is allocated a block of memory, as is each function. Each block is contiguous and is limited in size only by the size of memory available in the system. Dynamic memory allocation (*malloc* and *free*) can be implemented by declaring a large global or static array and allocating memory from it.

> **Note:**
>
> Placement of both code and data is done with the linker. Each item of code or data can be individually placed in memory, but generally this is not necessary. Memory-mapped I/O can be an exception, but access to such physical locations can usually be accomplished with C pointer types.

5.1.1 TMS34010 C Stacks

In TMS34010 C, there are two stacks: the program stack (STK) and the system stack (SP). The program stack passes parameters to functions and allocates local frames for functions. The system stack saves the status of the calling function; in other words, it saves registers. The two stacks are allocated by the boot module as a single static array called *sys_stack*. The program stack pointer STK is set to the bottom of this array (the low address) and grows up to higher addresses, while the system stack is set to the top of the array (the high address) and grows down to lower addresses. Thus, the two stacks grow toward each other, as illustrated in Figure 5-1. This arrangement must not be altered by customizing the boot module.

Three registers are reserved for stack management:

SP	Points to the top of the system stack
A14 (STK)	Points to the top of the program stack
A13 (FP)	Points to the beginning of the current frame (frame pointer)

Manipulation of these registers is done automatically for C functions by the C environment; however, assembly language routines linked with C functions must manipulate these registers according to the GSPC conventions.

5.1.2 Global Variable Memory Allocation

Each external or static variable declared in a C program is allocated an exclusive contiguous space in memory by the compiler. Whereas the actual address of the space is decided by the linker, the codegen provides that the space is always allocated in multiples of words, and each variable is always aligned on a word boundary.

5.1.3 Structure Packing and Field Manipulation

Structure elements are generally allocated space as needed to hold them. Fields are allocated as many bits as requested, enumerated types are allocated as few bits as possible to hold the maximum value of that type, bytes are allocated eight bits, and so on. See also TMS34010 C Data Sizes, page 4-3, and Enumeration Declarations, page 4-6.

In the TMS34010, structure mapping is done according to standard C practice with one exception: a field of declared width zero does not cause a word alignment. Because of the GSP's bit-addressability, word alignment in a structure does not necessarily produce more efficient code. However, note that fields which straddle word boundaries do take longer to access since both words must be fetched by the processor; thus, it is advisable to define structures and arrays of structures carefully to avoid fields which cross word boundaries.

If a structure is declared as an external or static variable, it is always placed on a word boundary and is allocated space rounded up to a word boundary. However, when an array of structures is declared, no rounding of size is used: exactly enough space is allocated to hold each structure element in contiguous bits of memory.

5.1.4 Array Alignment

In ANSI standard C, as well as K&R C (Kernighan and Ritchie), arrays are expected to always align their elements on a word boundary, with the exception of bytes, which may be aligned on a byte boundary. Because the TMS34010 is bit-addressable, this restriction becomes both unimportant and inefficient. Thus, in TMS34010 C, arrays have no internal alignment. Each element of the array is allocated exactly as much space as needed to hold the contents with no space between adjacent elements.

> **Note:**
>
> Like structures, a carefully defined array (with no elements overlapping word boundaries) will allow the program to run faster. In general, pixel arrays are so aligned.

If an array is declared as an external or static variable, the first element of the array is placed on a word boundary and the array is allocated space rounded up to a word boundary.

This method of handling an array allows more control over the environment than standard C allows. Arrays of bits or pixels are now directly accessible (a

necessity for a graphics environment), and memory-mapped I/O is much more straightforward.

5.2 Register Conventions

Strict conventions are used to determine which registers are used for what operations in the C environment. A good understanding of these conventions is necessary to interface assembly language to a C program.

5.2.1 Dedicated Registers

Three registers are used exclusively by the C environment and cannot be modified by a user in any way other than those prescribed in Section 5.5, Function Call Conventions, page 5-7. These registers are:

SP	Pointer to the top of the system stack
A14 (STK)	Pointer to the program (user) stack
A13 (FP)	Pointer to the base of the currently active frame

5.2.2 Using Registers

Registers A0 through A12 are generally available for use by a function. However, two conventions apply:

1) A function must save the contents of each register used on entrance to the function and restore those contents on exiting the function. A8 is the only exception; its contents do not have to be saved and restored.

2) If an integer value or a pointer is to be returned from a function, it must be placed in A8.

The codegen uses the registers in the following fashion:

Address generation	A0, A2, A4, A6
Expression analysis	A1, A3, A5, A7
Holding constants	A1, A3, A5, A7, A9 through A12
Return value/Scratch	A8
User register variables	A9, A10, A11, A12

Note:

Registers B0 through B14 are available for use by the assembly-language programmer and are not used by the C compiler.

Address generation registers are allocated on a least-recently-used (LRU) basis, with the low registers allocated first. Expression analysis registers are allocated from high to low registers, based on availability and current use. (Note that all integer expression analysis uses 32-bit math.) To hold constants, both the user registers and the expression analysis registers can be used and are allocated from high to low registers.

> **Note:**
>
> The codegen constantly keeps track of the contents of the registers and will attempt to reuse register data if it is at all possible. Therefore, it is inadvisable to modify with an in-line assembly construct (or with a modification of codegen output) any register already used in a function. Use the *-r* option of the codegen to produce the compiler's information about use of registers and to place it in the output file.

5.2.3 Register Variables

The codegen provides up to four active register variables at a time for each function. These are requested via the *register* storage class. (Refer to Storage Class Specifiers in Declarations, page 4-5, and K&R for more information.) The codegen allocates these variables from registers A9 through A12 in ascending order; thus, the first variable declared *register* is placed in A9, the second in A10, and so on. A register variable can contain any integer type, a pointer to any type, or a float (doubles or structures are not allowed). If more than four register variables are declared, the excess are treated as normal variables.

Register variables declared as *short* or *char* are treated as *long*.

> **Note:**
>
> Using register variables **vastly** increases the efficiency of code generated for some statements, sometimes by a factor of two or more. Because the codegen makes no attempt to keep track of operations involving register variables, you are free to manipulate them by using in-line assembly language.

5.3 Integer Expression Analysis

All integer expression analysis is performed in **A** file registers using the GSP 32-bit math instructions. Moreover, all multiplicative operations are performed into odd registers. For this reason, only A1, A3, A5, and A7 are used for general-purpose expression registers.

TMS34010 C follows exactly the standard precedence rules of K&R C. Order of analysis, however, for any operator's operands is based on the relative complexity of the operands: the more complex operand is always analyzed first. In this way, the codegen minimizes the number of operations which must be performed to fully analyze an expression. (This does not apply to operators which specify order of analysis of operands, such as the comma, &&, and || operators.)

If the codegen runs out of registers for use, one of these used registers is selected for reuse and its contents are saved on the system stack to be restored later. This frees a register temporarily.

5.4 Floating Point Conventions

A floating point value is represented in 32 bits for single precision and in 64 bits for double precision. All operations are done in double precision, so single precision (32-bit) values and integers are converted before any operations are performed. The following functionality is provided:

- Addition, subtraction, multiplication, division, negation, increment, decrement
- Comparisons
- Conversions
  - Unsigned integer to double, double to unsigned integer
  - Unsigned integer to float, float to unsigned integer
  - Signed integer to double, double to signed integer
  - Signed integer to float, float to signed integer
  - Float to double, double to float
- Error handling

The TMS34010 floating point package is a custom-coded package that does not follow usual C calling conventions. The calling conventions for routines work like a classic operand stack. First, the codegen pushes the floating point argument(s) onto the argument stack, then generates a call to a floating point function. The floating point function pops the arguments off the stack, performs the operation and pushes the result back onto the stack. The compiler has no knowledge of the internal format of the floating point numbers, and the only restriction is on the size of the number. This allows you to customize a floating point package for your environment.

Some floating point functions expect integer arguments or return integer values. For floating point functions, all integers are passed and returned in register A8.

The following functional definitions apply to the floating point package and are used by the compiler. More detailed information about each one of the functions can be found in Appendix D.

_FP_ADD, _FP_MINUS, _FP_DIV, _FP_MULT Each takes two doubles and returns a double result.

_FP_NEGATE Returns the negated value of the operand passed.

_FP_COMPARE Takes two floats and a comparison operator (in A8) and returns an integer result of 0 or 1 based on the comparison; also sets the status.

> **Note:**
>
> The compiler assumes that the status register is also appropriately set upon return from the _FP_COMPARE function.

_FP_DECR Takes one double and returns two: the top of the stack is the original argument decremented by one; the second item on the stack is the original argument (unchanged).

_FP_INCR	Takes one double and returns two: the top of the stack is the original argument incremented by one; the second item on the stack is the original argument (unchanged).
_FP_ITOD	Converts the signed integer argument (in A8) to a double return value.
_FP_UTOD	Converts the unsigned integer argument (in A8) to a double return value.
_FP_DTOI	Converts the double argument to a signed integer return value (in A8).
_FP_DTOU	Converts the double argument to an unsigned integer return value (in A8).
_FP_ITOF	Converts the signed integer argument (in A8) to a float return value.
_FP_UTOF	Converts the unsigned integer argument (in A8) to a float return value.
_FP_FTOI	Converts the float argument to a signed integer return value (in A8).
_FP_FTOU	Converts the float argument to an unsigned integer return value (in A8).
_FP_FTOD	Converts the float argument to a double return value.
_FP_DTOF	Converts the double argument to a float return value.
fp_error	Called when an exception occurs in one of the floating-point math routines. Takes one argument, the error number. These numbers are defined by the floating point package, as is the action of the function (see Appendix D). This function is not generally called from user code but by the floating point package. This function follows standard calling conventions, so it can be defined entirely in C.

5.5 Function Call Conventions

All function calls performed in the C environment follow a strict set of rules used by the compiler to avoid corruption of the run-time environment. The rules depend on whether arguments are to be passed to a function, whether the function returns a value, and the type of the return value (if any).

Figure 5-1 illustrates a function call.

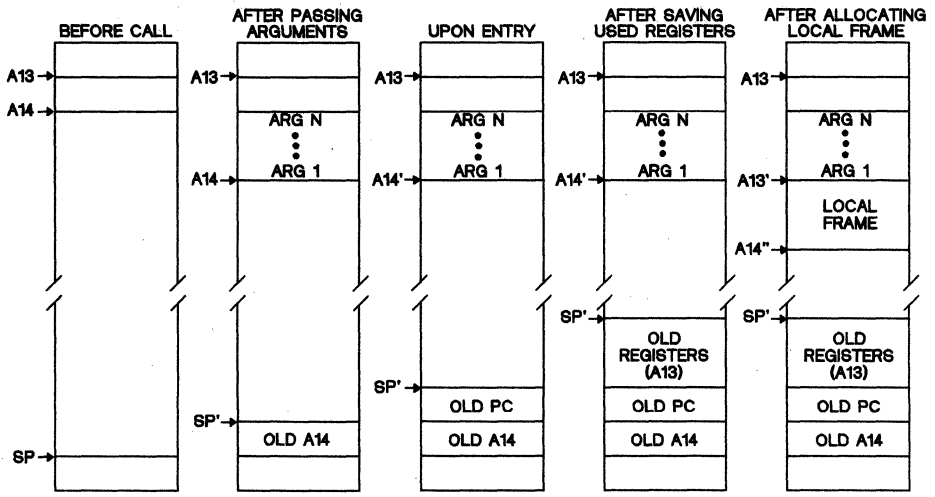

Figure 5-1. Typical Function Call with Parameters Passed, No Value Returned

5.5.1 Register Usage Within Functions

Each function must save the contents of each register used on entrance to the function and restore those contents on exiting the function. A8 is the only exception; its contents do not have to be saved and restored.

If an integer value or a pointer is to be returned from a function, it must be placed in A8.

Registers B0 through B14 are available for use by the assembly-language programmer and are not used by the C compiler.

5.5.2 Passing Parameters

Any parameters passed to a C function follow the strict rules used by the compiler. These rules are in the following list.

> **Note:**
>
> If these rules are not followed, the C run-time environment will be corrupted and will crash at some later time.

1) If the function is defined **not** to have any parameters, none may be passed.

2) If the function is defined to have one or more parameters, **at least one** parameter must be passed to the function.

3) If space must be allocated on the stack for a return value, it must be allocated before any other function call setup action.

   If the function returns a double, a float, or a structure type, space must be allocated on the stack for the return value, regardless of whether the caller will use the data.

4) Parameters to the function are passed in reverse order; that is, the last parameter is pushed first and the first is pushed last. This feature allows variable-length argument lists to be handled easily.

5) All integer types are widened to 32-bit integers. (See Conversions, page 4-4.)

6) All floating point values are converted to doubles.

7) Structures are "widened" to the next larger word boundary.

> **Note:**
>
> Passing structures to functions can be dangerous. The called function **must know** the correct size of the structure in order to handle the call properly. If the sizes do not match, unpredictable results occur. There is no compile-time check for this in C.

8) The called routine does all cleanup of the stack, so the caller does not need to do anything after the actual call to the function.

5.5.3 Local Frame Generation

The first steps taken by the called function are:

1) Saves any registers which are modified by the function (including the frame pointer if the function has local variables). This is accomplished via the MMTM instruction, pushing the original values of the used registers onto the system stack.

2) If there are local variables, sets the frame pointer (A13) to point to the top of the program stack (A14) and then allocates the local frame by incrementing the program stack.

Notes:

1. The status register is not saved.
2. There is not an explicit argument pointer in this scheme. The frame pointer (A13) is located at the end of the argument list and at the beginning of the local frame, so both arguments and local variables can be indexed from the frame pointer.
3. Register A8 is used as the return value register, and thus is never saved in a normal function. Therefore, this register can be modified without having to save it on the system stack.

5.5.4 Function Termination

At the termination of a function, the function:

1) Handles any return values to be passed to the caller.

2) Restores the caller's registers.

3) Restores the local frame of the caller.

There are three cases of return value types and each has a method of handling:

Pointers or integer return values: Returned in register A8. Because of this use of register A8, it is never saved at function entry. Thus, A8 may be used as a scratch register between function calls. Note that since integer values are not returned on the stack, it is not necessary for the caller to allocate space for them prior to passing parameters.

Floating point return values: Returned on the stack. Either a single or a double precision value can be returned. Upon return, the value appears to have been "pushed" on the stack. The space for the return value must be allocated by the caller before the call, followed by a save of the program stack pointer to the system stack.

Structure return values: Returned on the program stack, as floating point values are. It is assumed that the space is allocated, and that the program stack pointer (STK) is saved on the system stack.

Note:

The size of a structure returned is rounded up to the next word boundary. This makes handling easier. Upon return, the structure appears to have been "pushed" on the stack.

5.5.5 Restoration of the Caller's Environment

It is the responsibility of the called function to restore the environment of the caller before returning. In general, this involves restoration of the caller's registers and deletion of the local frame (including arguments).

The first task is to restore the caller's registers. This can be done with an MMFM instruction, corresponding to the MMTM given at function entry. Note that the status register need not be saved or restored. If local variables were allocated, the frame pointer must also be restored and should be included in the register list for the register save instructions.

There are two methods for deletion of local variables and arguments:

- Function returns value on stack, or has arguments. In the latter the old program stack pointer is pushed on the stack (below the old value). To restore the caller's stack, copy this value from the system stack (SP) to the program stack pointer, A14 (STK).

- No arguments, no return value on stack. If a local frame was allocated, decrement A14 by the size of the local frame and return. If no frame was allocated, just return.

Note that the old program stack pointer is not saved for either (1) the case of values returned on the stack or (2) the case of no arguments.

5.5.6 Return from Function

The last instruction executed by the function is a RETS. Normally, the instruction has an argument of zero, but if the function had arguments or returned a value on the stack, the instruction RETS 2 must be executed to pop the caller's old program stack pointer off the stack.

5.6 Interrupt Handling

Interrupts can be handled directly with C functions through the use of reserved function names. These names are of the form:

```
c_int##
```

where two pound signs (##) indicate a two-digit interrupt number. For example:

```
c_int00 (system reset interrupt)
c_int01 ... c_int99
```

By naming a function in this way, the user specifies that the function is to be used to handle an interrupt, and the codegen generates special code for an interrupt routine.

Any register used (with the exception of SP and STK) is saved by the interrupt handler, including A8. In a normal function, A8 need not be saved; however, in the case of an interrupt, A8 **must be saved**.

To return from the interrupt, use the RETI instruction. This restores the status of the interrupted function.

An interrupt function may perform any task a normal function may perform: access global variables, allocate local variables, call other functions, etc.

Notes:

1. It is your responsibility to handle any special masking of interrupts. You can reenable the interrupts and do any masking required without corrupting the C environment.
2. An interrupt handler must be declared with no arguments. If it is declared to have arguments, the interrupt handler will not run correctly.
3. An interrupt handler may not be called by the user code. Because the linkage is set up for handling interrupts, a C calling sequence will not be handled correctly, and the system could crash.
4. *c_int00* is used as the system reset interrupt. This routine initializes the system and calls the user's *main* function. Note that when the user's function is called, the interrupts are still disabled; thus, it is the user's responsibility to enable interrupts if they are needed.
5. Any interrupt handler can be used to handle any interrupt or multiple interrupts. The codegen does not generate any code specific to the particular interrupt, with the exception of the system reset interrupt *c_int00*, which must be used as system reset and cannot have any local variables (since it is assumed that at system reset the stack has not yet been allocated).

   To attach an interrupt handler to an interrupt, the address of the interrupt must be placed in the proper interrupt vector. This can be done with the assembler and linker, creating a simple table of addresses and linking to the proper location.

5.7 System Initialization

Before any C code may be run, the C run-time environment must be created. This environment is represented in the program stack and the system stack, so these must be properly constructed and initialized. In addition, any variables which were declared to be automatically initialized at program entry must be initialized before calling user code.

Two stacks are used to manage the C run-time environment:

Program stack (STK)
System stack (SP)

These stacks must be initialized by the boot function before any user code is executed. The stacks are located in the static array SYS_STACK, declared in the module BOOT.C, and share the space by growing toward each other from opposite ends of the space. A stack overflow occurs when the stacks overlap. The size of SYS_STACK may be changed by modifying source code and recompiling BOOT.C. There is no protection in the run-time code against stack overflows, so stack size should be chosen with care.

5.7.1 System Stack

The system stack is pointed to by the SP register and grows toward low memory. This stack is used by function calls to save the context of the calling function, including any registers used by the called function. This stack is supported directly by the TMS34010 instruction set, and register SP is dedicated to stack management. The system stack is manipulated by C primarily through the use of four commands:

- MMTM Save Registers
- MMFM Restore Registers
- CALLA (or CALL) Call a Function
- RETS (or RETI) Return from a Function (or Interrupt)

It is also used by interrupts to save the status of the interrupted function.

5.7.2 Program Stack

The program stack is used for frame generation; i.e., to pass arguments to functions and allocate local (temporary) variables for the called functions. The program stack is controlled entirely in software, using A14 to point to the current "top of stack", and grows toward high memory. Thus, A14 is a dedicated register in the C environment, and it must be carefully manipulated to avoid system crashes.

5.7.3 Initialization of Global Variables

Before execution of your program, any global variables declared to be preinitialized must be initialized by the boot program. This is done through the use of initialization tables placed in the *.data* section of the program object module. Any module generated by the compiler may produce these tables, and the linker appends them into one table.

> **Note:**
>
> Because the *.data* section is used by the codegen to contain initialization tables only, you are not allowed to place any other data in this section. Doing so causes corruption in the initialization table format, causing unpredictable results.

Global and static variables that are not autoinitialized are not guaranteed to be initialized to 0.

6. TMS34010 Run-Time Support

6.1 Memory Management

Five routines are provided to implement C dynamic memory management. These routines are:

malloc Allocates an area of a specified size in memory.

calloc Allocates and clears an area of memory.

realloc Re-allocates a previously allocated memory area with a new size.

free De-allocates space allocated by *calloc*, *malloc*, or *realloc*.

movmem Moves a specified number of bytes from one address to another.

6.1.1 Specifying the Size of Memory to Manage

The amount of memory managed by the memory management routines is specified by the macro *memory_size* in the RTS (run-time support) module *memory.c*, which contains the source code for all the memory management routines. By modifying this value and recompiling the module, the amount of memory used can be changed.

6.2 String Functions

Several functions are provided to allow manipulation of strings, search for characters, and comparison of strings. These functions are:

strcat * Appends a string onto the end of another string.

strncat * Appends a string of up to *n* characters onto another string.

strchr Searches for the first occurrence of a character in a string.

strrchr Searches for the last occurrence of a character in a string.

strcmp Compares two strings.

strncmp Compares up to *n* characters in two strings.

strcpy * Copies a string to a new location.

strncpy * Copies up to *n* characters of a string to a new location.

strlen Returns the length of a string.

* When using functions that move or copy strings, ensure that the destination is large enough to contain the result.

6.3 Character Typing and Conversion Macros

These macros determine the types of characters contained in variables, arrays, constants, etc. The library file *ctype.h* must be included in your file to use the functions. The character typing macros are:

isalnum Detects alphanumeric ASCII characters.

isalpha Detects alphabetic ASCII characters.

isascii Detects ASCII characters.

iscntrl Detects control characters.

isdigit Detects numeric characters.

islower Detects lowercase alphabetic ASCII characters.

isprint Detects printable ASCII characters.

ispunct Detects ASCII punctuation characters.

isspace Detects ASCII spacebar, tab (horizontal or vertical), carriage return, formfeed, and newline characters.

isupper Detects uppercase ASCII alphabetic characters.

isxdigit Detects hexadecimal digit characters.

Conversion of uppercase characters to lowercase, and vice versa, and conversion of non-ASCII characters to ASCII can be done with the following macros. Include the library file *ctype.h* to use them.

tolower Converts uppercase alphabetic characters to lowercase.

toupper Converts lowercase alphabetic characters to uppercase.

toascii Converts non-ASCII characters to ASCII characters.

6.4 Miscellaneous Functions

setjmp Saves calling function's context in environment buffer.

longjmp Restores context of calling function from environment buffer.

Syntax

```
double atof(nptr)
  char *nptr;
```

Description

This function converts a string of ASCII characters to floating-point values. The string is given in the format:

[space][sign]digits[.digits][e|E[sign]integer]

The *space* shown in the format is white space and is indicated by a space-bar, horizontal or vertical tab, carriage return, form feed, or newline. Following the space indicator is an optional *sign*, and then *digits* representing the integer part of the number. The fractional part of the number follows, then the exponent, including the option of a *sign*.

The first unrecognized character terminates the string.

The *atof* function does not account for any overflow resulting from the conversion.

Syntax

```
double atoi(nptr)
  char *nptr;
```

Description

This function converts a string of ASCII characters to integer values. The string is given in the format:

[space][sign]digits

The *space* shown in the format is white space and is indicated by a space-bar, horizontal or vertical tab, carriage return, form feed, or newline. Following the space indicator is an optional *sign*, and then digits representing the number.

The first unrecognized character terminates the string.

The *atoi* function does not account for any overflow resulting from the conversion.

Syntax

```
#include <memory.h>
char *calloc(num,size)
  int num; /* number of items to clear */
  int size; /* size of each item */
```

Description

This routine allocates a packet of memory large enough to contain *num* objects of the specified size and returns a pointer to it. If it cannot allocate the packet (i.e., if it runs out of memory), it returns a null pointer (0). This function also initializes the allocated memory to all zeros. Refer to Section 6.1.1, Specifying the Size of Memory to Manage, for more information on allocating memory.

Example

```
ptr = calloc(10,2) ;  /* allocate/clear 10 words */
```

Syntax

```
#include <memory.h>
int free(pointer)
  char *pointer;
```

Description

This routine deallocates a packet of memory (pointed to by *pointer*) previously allocated by *malloc*, *calloc*, or *realloc*. If you attempt to free a packet not previously allocated, the function takes no action and returns. Refer to Section 6.1.1, Specifying the Size of Memory to Manage, for more information on allocating memory.

Example

```
 char *x;
x = malloc(10);      /*  allocate 10 bytes   */
free(x);             /*  free 10 bytes       */
```

Syntax

```
#include <ctype.h>
int isxxxxx(character)
char character;
```

Description

These macros identify a particular type of character, such as alphabetic, alphanumeric, numeric, ASCII, etc. If the argument *character* is one of the characters designated by the macro name, the macro returns a nonzero integer. Otherwise, it returns 0. The character typing macros are:

isalnum	Detects alphanumeric ASCII characters.
isalpha	Detects alphabetic ASCII characters.
isascii	Detects ASCII characters.
iscntrl	Detects control characters.
isdigit	Detects numeric characters.
islower	Detects lowercase alphabetic ASCII characters.
isprint	Detects printable ASCII characters.
ispunct	Detects ASCII punctuation characters.
isspace	Detects ASCII spacebar, tab (horizontal or vertical), carriage return, formfeed, and newline characters.
isupper	Detects uppercase ASCII alphabetic characters.
isxdigit	Detects hexadecimal digit characters.

Syntax

```
#include <setjmp.h>
longjmp(env,val)
jmp_buf env; /* environment save buffer */
int val;     /* value to be returned by corresponding "setjmp" */
```

Description This function restores the context of a calling function's environment. It is used in conjunction with *setjmp*. These two functions provide transfer of control from a nested series of functions back to a specified point without using a series of *return* statements.

Syntax

```
int ltoa(n,buffer)
  long n;      /* number to convert */
  char *buffer;  /* buffer to put result in */
```

Description This function converts a long integer to the equivalent ASCII string. If the input number is negative, a leading minus sign is output. The *ltoa* function returns the number of characters placed in the buffer.

Syntax

```
#include <memory.h>
char *malloc(size)
 int size;  /* size of block in bytes */
```

Description

This routine allocates a packet of memory of a specified size and returns a pointer to it. If *malloc* is unable to allocate the packet (i.e., if it runs out of memory), it returns a null pointer (0). This function does not modify the memory it allocates. Refer to Section 6.1.1, Specifying the Size of Memory to Manage, for more information on allocating memory.

Syntax

```
#include <memory.h>
char *movmem(src,dest,count);
  char *src ; /* source address  */
  char *dest; /* destination address  */
  char count; /* number of bytes to move */
```

Description

This routine moves *count* bytes of memory from *source* address to *dest* address. The source and destination areas can be overlapping. Refer to Section 6.1.1, Specifying the Size of Memory to Manage, for more information on allocating memory.

Syntax

```
#include <memory.h>
char *realloc(ptr,newlength)
  char *ptr ; /* number of items to clear */
  int newlength ; /* new size of packet */
```

Description

This routine changes the size of the allocated data area pointed to by the first argument, *ptr,* to the size specified by the second argument, *newlength*. It returns a pointer to the space allocated since the packet and its contents may have to be moved in order to expand. Any memory freed by this operation is deallocated. If an error occurs, the function returns zero. Refer to Section 6.1.1, Specifying the Size of Memory to Manage, for more information on allocating memory.

Syntax

```
#include <setjmp.h>
int setjmp(env)
jmp_buf env;
```

Description

This function saves the context of a calling function's environment. It is used in conjunction with *longjmp*. These two functions provide transfer of control from a nested series of functions back to a specified point without using a series of *return* statements.

The *setjmp* function returns zero when called, but if *longjmp* is executed it "returns" a second time, returning the value specified by the *longjmp*.

Syntax

```
#include <string.h>
char *strcat(string1,string2)
char *string1,*string2;
```

Description The *strcat* function appends a copy of *string2* to the end of *string1* and returns a pointer to the first character of *string1*

Syntax

```
#include <string.h>
unsigned char *strchr(s,c)
unsigned char *s,c;
```

Description

This function finds the first occurrence of the character *c* in the string *s* and returns a pointer to it. If the character is not found, *strchr* returns a null.

strchr is equivalent to other C compilers' *index*.

Syntax

```
#include <string.h>
int strcmp(string1,string2)
char *string1,*string2;
```

Description This function compares two strings, character by character, and returns an indicator of which string is lower in the ASCII character set. The indicators are:

-1	if *string1* is less than *string2*
0	if *string1* is equal to *string2*
1	if *string1* is greater than *string2*

Syntax

```
#include <string.h>
char *strcpy(to,from)
char *to,*from;
```

Description This function copies the string at the address *from* into the address *to* and returns a pointer to this destination string.

Syntax

```
#include <string.h>
unsigned int strlen(string)
char *string;
```

Description The function returns the length of *string*. In C, a character string is terminated by the first byte with a value of zero. The returned result does not include the zero byte.

Syntax

```
#include <string.h>
char *strncat(string1,string2,max)
char *string1,*string2;
unsigned int max;
```

Description

This function appends a copy of *string2*, or the first *max* characters of *string2* (whichever is smaller), to the end of *string1*. It also returns a pointer to the destination string.

Syntax

```
#include <string.h>
char *strncmp(string1,string2,n)
char *string1,*string2;
unsigned int n;
```

Description This function compares two strings, character by character, and indicates which string is lower in the ASCII character set. Only *n* characters are compared; if either string has less than *n* characters, the comparison stops at the end of the shorter string. The indicators are:

-1	if *string1* is less than *string2*
0	if *string1* is equal to *string2*
1	if *string1* is greater than *string2*

Syntax

```
#include <string.h>
char *strncpy(to,from,n)
char *to,*from;
unsigned int n;
```

Description

This function copies the string at address *from* into the address *to*. Only *n* characters are copied. If the input string is less than *n* characters in length, the remainder of the destination field is padded with zeros. If the input string has *n* or more characters, the string *to* **will not be terminated by an end of string character**. The *strncpy* function returns the address of the destination string.

Syntax

```
#include <string.h>
unsigned char *strrchr(s,c)
unsigned char *s,c;
```

Description

This function finds the last occurrence of the character *c* in string *s* and returns a pointer to it. If the character *c* is not found in the string, *strrchr* returns a null.

strrchr is equivalent to other C compilers' *rindex*.

Syntax

```
#include <ctype.h>
char toascii(c)
char c;
```

Description This macro converts the argument *c* to its ASCII equivalent by masking the bottom seven bits.

Syntax

```
#include <ctype.h>
char tolower(c)
char c;
```

Description This macro converts the uppercase argument *c* to its lowercase form. If the argument is already in lowercase, the macro returns the argument unchanged.

Syntax

```
#include <ctype.h>
char toupper(c)
char c;
```

Description

This macro converts the lowercase argument *c* to its uppercase form. If the argument is already in uppercase, the macro returns the argument unchanged.

A. Fatal Errors

Compiler error messages appear in a straightforward format showing the line number in which the error occurs and the text of the message:

"name.c", line n: <error message>

Any of the errors shown in this section cause the compiler to abort immediately. Text enclosed in single or double quotes in these error messages is replaced with actual text from the program, your own symbols, filenames, memory allocations, etc.

error: cannot allocate sufficient memory

The compiler requires a minimum of 512K bytes of memory to run; this message indicates a lack of necessary memory. Supply more dynamic RAM.

error: can't open "filename" as source

The compiler cannot find the file name as entered. Check for spelling errors; check for the existence of the file named.

error: can't open "filename" as intermediate file

The compiler cannot create the output file. This is usually caused by either an error in the syntax of the filename or a full disk.

error: illegal extension "ext" on output file

The intermediate file cannot have a ".c" extension.

*****fatal errors found: no intermediate file produced

This message is printed after an unsuccessful compilation. Correct the errors (other messages will indicate particular errors) and try compilation again.

cannot recover from earlier errors: goodbye!

An error has occurred that prevents the compiler from continuing.

B. Reference Documents

The following Texas Instruments publications provide additional information about the TMS34010 and may be ordered from a TI Sales Office or authorized distributor.

- *TMS34010 User's Guide*, (SPVU001)
- *TMS34010 Assembly Language Tools User's Guide*, (SPVU004)
- TMS34010 Data Sheet (SPPS011)
- TMS4161 Data Sheet
- TMS4461 Data Sheet

The following books or articles provide further background in C programming, in graphics, and system concepts associated with graphics.

- Cody, William J., Jr., and William Waite, *Software Manual for the Elementary Functions*, Prentice Hall, Englewood Cliffs, New Jersey, 1980.
- Kernighan, B, and D. Ritchie. *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
- Kochan, Steve G. *Programming in C*, Hayden Book Company.
- Sobelman, Gerald E., and David E. Krekelberg. *Advanced C: Techniques and Applications*, Que Corporation, 1985.
- Harbison, S., and G. Steele. *C: A Reference Manual*, Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

C. C Preprocessor Directives

The C preprocessor provided with this package is standard and follows K&R exactly. This appendix is simply an overview of the functionality of the directives.

Syntax #define <name>(arg,...,arg) <token-string>

Description Subsequent occurrences of *name* followed immediately by a list of arguments separated by commas and enclosed in parentheses are replaced by *token-string*, where each occurrence of an argument is replaced by the corresponding set of tokens from the comma-separated string. When a macro with arguments is expanded, the arguments are placed into the expanded *token-string* unchanged. After the entire *token-string* has been expanded, GSPCPP scans again for names to expand at the beginning of the newly created *token-string*, which allows for nested macros.

Note that there is no space between *name* and the open parenthesis at the beginning of the argument list. Also, there is no trailing semicolon (;).

Example

```
#define f(a,b,c) 3*a+b-c
```

causes

```
f(27,begin,minus)
```

whenever it occurs in the code, to be expanded to

```
3*27+begin-minus
```

Syntax #else

Description The lines of code between this directive and *#endif* will appear in the output if and only if the test directive corresponding to this *#else* produces an untrue result.

Syntax #endif

Description Ends a section of lines begun by a test directive (*#if*, *#ifdef*, or *#ifndef*). Each test directive must have a matching *#endif*. Conditional compilation sequences may be nested.

Syntax #if <constant-expression>

Description The lines of code between #if and *#endif* or *#else* will appear in the output if and only if the *constant-expression* evaluates to a non-zero value. All binary non-assignment C operators, the ?: operator, the unary -,], and ! operators are all legal in *constant-expression*. The precedence of the operators is the same as in the definition of the C language. There is also a unary operator called *defined*, which can be used in *constant-expression* in these two forms:

defined(<name>), or

defined <name>

This allows the the utility of *#ifdef* and *#ifndef* in a *#if* directive. Only these operators, integer constants, and names which are known by GSPCPP should be used in *constant-expression*. In particular, the *sizeof* operator is not available.

Syntax #ifdef <name>

Description Inserts the lines of code between *#ifdef* and *#endif* or *#else* into the output if and only if *name* has been defined (by *#define*) and has not been the subject of a subsequent *#undef*.

Syntax #ifndef <name>

Description The lines of code between *#ifndef* and *#endif* or *#else* will appear in the output if and only if *name* has not been defined (by *#define*) or has been the subject of a subsequent *#undef*.

Syntax

#include "filename"
or
#include <filename>

Description

Includes at this point in the code the contents of the given *filename*, which will then be run through GSPCPP. Either double quotes or angle brackets are used to enclose *filename*.

The filename can include a pathname with directories and so forth. If no directory is specified, the file is looked for in:

1) Current directory
2) Directories specified using -*I* option (see Section 3.1.3, Operation of GSPCPP, page 3-3).

Syntax #line <integer-constant> ["<filename>"]

Description Causes generation of line control information for the next pass of the compiler. *integer-constant* is the line number of the next line, and *filename* is the file where that line exists. If no filename is given, the current filename (given by the last *#line* directive) is unchanged.

This directive can be used to set the __LINE__ and __FILE__ symbols (see Section 3.1.3, Operation of GSPCPP).

Syntax #undef <name>

Description Causes the definition of *name* to be forgotten.

D. Floating Point Facility

The TMS34010 C floating-point package allows you to perform real number arithmetic operations from C. The TMS34010 floating-point functions have a special C assembly language interface and, therefore, do not adhere to the C function calling convention. The TMS34010 C compiler performs real number arithmetic in double precision. The following operations are included in the C floating point package:

- Addition, subtraction, multiplication, division, negation, increment, and decrement for double-precision format
- Comparisons for double precision format
- Conversions
  - Signed/unsigned integer to double
  - Double to signed/unsigned integer
  - Signed/unsigned integer to float
  - Float to signed/unsigned integer
  - Float to double
  - Double to float
- Creation of a floating point number in either single- or double-precision format from an ASCII string
- Error detection and exception handling.

These operations/conversions are described on the following pages.

D.1 Single-Precision Floating-Point Format

The single-precision floating-point format supported by the TMS34010 C floating-point RTS is a 32-bit format: a sign bit, an 8-bit biased exponent, and a 23-bit mantissa. The three component fields are positioned in the 32-bit field format as follows:

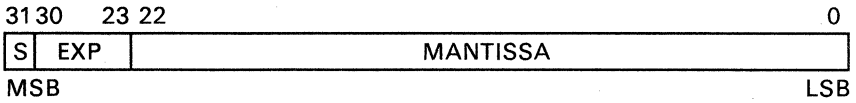

Figure D-1. Single-Precision Floating Point Format

Hence, the sign is in bit 31 of the 32-bit field, the exponent resides in bits 23 through 30, and the mantissa in bits 0 through 22.

Given a sign bit *s*, an exponent *e*, and a mantissa *f*, then the value V of the floating point number X=(*s,e,f*) is as follows:

- If s = 0, e = 255, and f = 0, then V = +infinity
- If s = 1, e = 255, and f = 0, then V = -infinity
- If $0 < e < 255$ and $f \neq 0$, then $V = (-1)^{s} * 2^{e-127}(.f)$
  (V = Not Valid if the number X=(s,e,f) is not a normalized floating point number. See Floating Point Arithmetic, Rounding, and Normalization, page D-4.)
- If s = 0, e = 0, and f = 0, then V = 0
- For all other cases, V = Not Valid

Precision in the single-precision format is greater than six decimal digits and the range includes the following:

- 5.87747×10^{-39} to 1.70141×10^{38} (positive range)
  (5.87747 E-39 to 1.70141 E38, in C format)
- $-1.70141 * 10^{38}$ to $-5.87747 * 10^{-39}$ (negative range)
  (-1.70141 E38 to -5.87747 E-39, in C format)
- 0 (zero)
- +/- infinity

D.2 Double-Precision Floating-Point Format

The double precision floating point format supported by the TMS34010 C floating point RTS is a 64-bit format: a sign bit, an 11-bit biased exponent, a 52-bit mantissa. The three component fields are positioned in the 64-bit field format as follows:

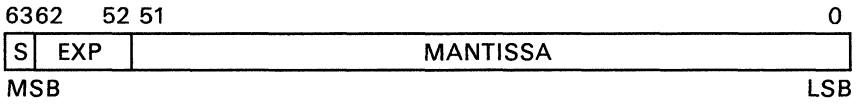

Figure D-2. Double-Precision Floating Point Format

Hence, the sign bit is bit 63 of the 64-bit field, the exponent resides in bits 52 through 62, and the mantissa is in bits 0 through 51.

Given an sign bit *s*, an exponent *e*, and a mantissa *f*, then the value V of the floating point number X=(*s,e,f*) is as follows:

- If s = 0, e = 2047, and f = 0, then V = +infinity
- If s = 1, e = 2047, and f = 0, then V = -infinity
- If $0 < e < 2047$ and $f \neq 0$, then $V = (-1)^s 2^{e-1023}(.f)$
  (V = Not Valid if the number X=(s,e,f) is not a normalized floating point number. See Floating Point Arithmetic, Rounding, and Normalization, page D-4.)
- If s = 0, e = 0, and f = 0, then V = 0
- For all other cases, V = Not Valid

Precision in the double- precision format is greater than fifteen decimal digits and the range includes the following:

- 1.11254×10^{-308} to 8.98847×10^{308} (positive range)
  (1.11254 E-308 to 8.98847 E308, in C format)
- -8.98847×10^{308} to $-1.11254 \times 10^{-308}$ (negative range)
  (-8.98847 E308 to -1.11254 E-308, in C format)
- 0 (zero)
- +/- infinity

D.3 Floating Point Arithmetic, Rounding, and Normalization

While floating point packages do not always implement a rounding scheme, rounding does increase the precision of operations and conversions. The TMS34010 C floating point RTS package implements rounding in any function where it is applicable. The particular scheme implemented in the TMS34010 RTS package is known as "round toward infinity". This is the natural rounding method whereby negative-valued floating point numbers are rounded toward negative infinity and positive-valued numbers are rounded toward positive infinity. Because all mantissas are unsigned, this type of rounding is accomplished by adding one to the retained part of a mantissa whenever the most significant bit discarded was a one. The exception to this is when a floating point number is converted to an integer, in which case the fractional part is **truncated**.

In addition to the "round toward infinity" feature, the TMS34010 C floating point RTS package also implements extended real-number arithmetic. This means that the subset of real numbers representable by the RTS package, along with +/- infinity, form a well-ordered set. Moreover, it means that arithmetic operations with +/- infinity as one or both operands can be performed with certain error generating exceptions (see Table D-1, page D-7). The extended real number system is critical to advanced mathematic and engineering disciplines and its implementation can, therefore, be extremely useful.

> **Note:**
>
> The implementation of extended real-number arithmetic does not affect the arithmetic when neither operand is +/- infinity. Extended real arithmetic is strictly an added feature.

Floating-point numbers which have an invalid value V are never generated by TMS34010 C floating point functions. In addition, all floating point numbers generated by the TMS34010 C functions are normalized. Denormalized numbers are considered to be invalid and as such are never generated by any function.

Normalized floating-point numbers (with the exception of zero and +/- infinity) have a one in the MSB of the mantissa. Denormalized results are always normalized before their return. If normalization cannot be accomplished in the format's range, then the appropriate underflow error is generated. A floating-point number is normalized by shifting the mantissa left until the MSB of the mantissa is one and then subtracting the shift count from the exponent. An example of single precision normalization is shown in Figure D-3.

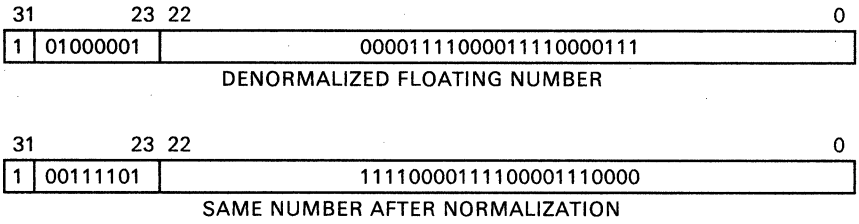

Figure D-3. Single-Precision Normalization

Normalization for the double precision format is done in the same manner.

D.4 Floating Point Interface

Floating point functions do not adhere to the C calling convention and cannot be called via a C function call. Each floating point function requires its arguments to be in a particular order on the program stack: the 32 least-significant bits of double precision numbers are pushed onto the stack first; the 32 most-significant bits are pushed on last. Integer arguments and results are passed via register A8.

If you wish to call the functions directly (bypassing the C compiler), you must do so via TMS34010 assembly language.

D.4.1 Floating Point Conversions

The TMS34010 C floating-point conversion functions can be classified into two categories:

- The "exact" conversion functions:

  _FP_ITOD - signed integer to double
  _FP_FTOD - float to double
  _FP_UTOD - unsigned integer to double

- The "best fit" conversion functions:

  _FP_DTOF - double to float
  _FP_FTOI - float to signed integer
  _FP_DTOI - double to signed integer
  _FP_ITOF - signed integer to float
  _FP_UTOF - unsigned integer to float

The "exact" conversion functions are such that the value V_s of the number in the source format is exactly equal to V_d, the value of the number in the destination format. Hence, "exact" conversion functions never cause a loss of precision, and a "faithful" representation of the source number in the destination format is one that has the same value ($V_s = V_d$).

The "best fit" conversion functions are those which cannot always make the conversion so that $V_s = V_d$. Hence, they must convert to the closest possible

value representable in the destination format. Since the set of possible values representable by the destination format is always finite (although very large), this "best fit" value V_d does exist and is unique. A conversion error is generated whenever V_s lies outside the range of the destination format unless $-1 < V_s < 1$. A "faithful" representation of the source number in the destination format is defined as the unique number such that V_d best fits V_s.

D.5 Floating-Point Error Exception Handling

Numerous errors are detectable by the TMS34010 C floating-point run-time support functions, and you can trap every error that is detected. Each error can be trapped independently, so you are free to trap one type of error while accepting the default result on any or all others. This trapping is done by means of a user-definable error exception handling routine called fp_error. This routine is called by the floating point package whenever one of the detectable errors occurs. For a complete list of detectable errors, see Table D-1, page D-7. When an error is encountered, fp_error is called using the C calling convention, with an error number passed as an argument.

An example of *fp_error* might be:

```
void fp_error(err) int err;
{  switch(err)
   {
     1:
     2:
     3: printf("CONVERSION ERROR"); return;
     5:
     6:
     7: printf("ARITHMETIC OVERFLOW"); return;
   }
}
```

Note:

This example routine attempts no error recovery, but it would be useful in debugging an algorithm.

Table D-1. Floating Point Error Descriptions

Error Code (decimal)	Functions Generating the Error	Error Description
1 2 3	_FP_DTOI _FP_DTOU _FP_DTOF _FP_FTOI _FP_FTOU	Conversion error: Generated if destination format cannot faithfully represent source value(essentially an overflow occurs when trying to make the conversion from source format to destination format).
4	_FP_ADD	Infinity - infinity: Default result is zero.
5 6 7	_FP_ADD _FP_MUL _FP_DIV	Overflow: Generated when there is an overflow during an arithmetic operation (i.e., the result is too big to be represented in the given format). Default result is +/- infinity.
11	_FP_MUL	Infinity x 0: default result is zero.
12 13 14	_FP_ADD _FP_MUL _FP_DIV	Underflow: Generated when there is an underflow during an arithmetic operation (i.e., the result is too small to be represented in the given format). Default result is zero.
15	_FP_DIV	Divide by zero: Default result is +/- infinity.
16	_FP_DIV	Infinity/infinity: Default result is +/-1.

D.5.1 Default Error Exception Handler

The default error exception handler *fp_error* can be modified to meet your specific needs. As supplied, the routine is as follows:

```
void fp_error (error_code)
int error_code;
{
 return; /*  does nothing  */
}
```

D.6 Function Descriptions

This section contains a description of each of the TMS34010 C floating point RTS (Run-Time Support) functions. Each function is covered in a one-page summary describing its inputs, outputs, error detection, and argument passing. Argument passing is described via a convention indicating arguments as described below. The following notations and conventions are used in the function summaries:

float	single-precision floating-point number (32 bits)
double	double-precision floating-point number (64 bits)
int	integer (32 bits)
opcode	32-bit operation code

Note:

There are never more than two inputs to a function, with the exception of FP_COMPARE, which has three. Also, every function returns only one result though it may be an integer, a single precision, or a double precision number. Remember that single and double precision results are returned via the argument stack, while integer results are returned via register A8.

Input(s) double1, double2

Output(s) double

Action double1 + double2 → double

Description This function takes two double-precision floating-point numbers and forms their sum.

Errors Overflow (error code # = 5)
Underflow (error code # = 12)
Infinity - Infinity (error code # = 4)

Example

This is an example of C code which generates a call to _FP_ADD.

```
main()
{
 double  a,b,c ;      /* declare three double
                         precision floating
                         point numbers       */
 c = a + b ;
};                     /* end of main         */
```

Input(s) double1, double2, op

Output(s) int

Action If (double1 op double2), then 1 → int
Else 0 --> int
where op is defined as follows:

1 == equal
2 |= not equal
3 <= less than or equal
4 < less than
5 >= greater than or equal
6 > greater than

Description This function takes two double-precision floating-point numbers and compares them according to a third input, *op*. The integer result is zero if the comparison is false and one if it is true. The inputs *double1* and *double2* are compared according to *op*, with *double1* being the leftmost argument.

Errors None.

The subset of extended real numbers representable by the TMS34010 double-precision floating-point format forms a well ordered set.

Example

The following is an example of C code which would generate a call to _FP_COMPARE.

```
main()
{
  double  a,b ;        /* declare two double
                          precision floating
                          point numbers          */
  if (a < b) ;         /* OP = 2 when _FP_COMPARE
                       is called                 */
}                      /* end of main              */
```

Input(s) double1

Output(s) double1, double2

Action double1 - 1 → double2

Description This function takes a double-precision floating-point number and decrements it by one. The original input is left untouched as the second item on the stack. The result is at the top of the argument stack after execution of this function.

Errors None.

There are no errors generated in _FP_DECR because _FP_DECR merely pushes a double-precision binary floating-point one onto the argument stack and calls _FP_MINUS. Errors are therefore detected as in _FP_MINUS.

Example

The following is an example of C code which would generate a call to _FP_DECR.

```
main()
{
 double  a ;          /* declare one double
                         precision floating
                         point numbers          */
 a--;
}                     /* end of main             */
```

Input(s) double1, double2

Output(s) double

Action double2 / double1 → double

Description This function takes two double-precision floating-point numbers and forms their quotient. The remainder is not returned.

Errors Overflow (error code # = 7)
Underflow (error code # = 14)
Divide by zero (error code # = 15)
Infinity / Infinity (error code # = 16)

Example

The following is an example of C code which would generate a call to _FP_DIV.

```
main()
{
 double  a,b,c ;        /* declare three double
                           precision floating
                           point numbers          */
 c = a / b ;
}                       /* end of main             */
```

Input(s) double

Output(s) float

Action double → float

Description This function converts a double-precision floating-point number to single-precision floating-point format.

Errors Conversion Error (error code # = 2)

The conversion error is generated only when the value V of the double precision number (after rounding) is too big to be faithfully represented in the single precision format.

Example

The following is an example of C code which would generate a call to _FP_DTOF.

```
main()
{
 double  a ;     /* declare one double and one */
 float b ;       /* single-precision binary
                    floating-point number    */
 b = a ;
}                /* end of main              */
```

Input(s) double

Output(s) int

Action double → int

Description This function converts a double-precision floating-point number to a signed 32-bit integer format.

Errors Conversion Error (error code # = 1)

The conversion error is generated only when the value V of the double precision number (after truncation) is too big to be faithfully represented in the signed 32-bit binary integer format.

Example

The following is an example of C code which would generate a call to _FP_DTOI.

```
main()
{
 double  a ;          /* declare one double- */
 integer i ;          /* precision floating-
                      point number and one
                      integer  */

 i = a ;
}                     /* end of main          */
```

Input(s) double

Output(s) unsigned int

Action double → unsigned int

Description This function converts a double-precision floating-point number to an unsigned 32-bit integer format.

Errors Conversion Error (error code # = 1)

The conversion error is generated only when the value V of the double precision number DI1 (after truncation) is too big to be faithfully represented in the signed 32-bit binary integer format.

Example

The following is an example of C code which would generate a call to _FP_DTOU.

```
main()
{
 double  a ;          /* declare one double- */
 unsigned j ;          /* precision floating-
                       point number and one
                       integer  */
 j = a ;
}                      /* end of main          */
```

Syntax void fp_error(err)
int err;

Input(s) integer

Output(s) User defined. (Default = no output)

Action User defined. (Default = no action)

Description This function is a user-definable error exception handler. The default function provided with the TMS34010 C floating point package does nothing more than pop the input off the program stack and return to the calling program.

Errors User defined. (Default = none)

Input(s) float

Output(s) double

Action float → double

Description This function converts a single-precision floating-point number to double-precision floating-point format.

Errors None.

Example

The following is an example of C code which would generate a call to _FP_FTOD.

```
main()
{
 double  a ;          /* declare one double and one
 float b ;            /* single precision binary
                       floating point number   */
 a = b ;
}                     /* end of main             */
```

Input(s) float

Output(s) int

Action float → int

Description This function converts a single-precision floating-point number to signed 32-bit integer format.

Errors Conversion Error (error code # = 3)

The conversion error is generated only when the value V of the double-precision number (after truncation) is too large to be faithfully represented in the signed 32-bit integer format.

Example

This example of C code generates a call to _FP_FTOI.

```
main()
{
 float   a ;          /* declare one single   */
 integer i ;          /* precision floating
                      point number and one
                      integer  */
 i = a ;
}                     /* end of main       */
```

Input(s) float

Output(s) unsigned int

Action float → unsigned int

Description This function converts a single-precision floating-point number to unsigned 32-bit integer format.

Errors Conversion Error (error code # = 3)

The conversion error is generated only when the value V of the double precision number (after truncation) is too large to be faithfully represented in the signed 32-bit integer format.

Example

This example of C code generates a call to _FP_FTOU.

```
main()
{
 float    a ;          /* declare one single  */
 unsigned j ;          /* precision floating
                       point number and one
                       integer  */
 j = a ;
}                      /* end of main       */
```

Input(s) double1

Output(s) double1, double2

Action double1 + 1 → double2

Description This function takes a double-precision floating-point number and increments it by one. The original input is not popped off the stack but is left untouched as the second item on the stack. The result is at the top of the program stack after execution of this function.

Errors None.

There are no errors generated in _FP_INCR because the function merely pushes a double-precision floating-point number onto the program stack and calls _FP_ADD. Errors are therefore detected as in _FP_ADD.

Example

This example of C code generates a call to _FP_INCR.

```
main()
{
 double  a ;          /* declare one double
                         precision floating
                         point numbers              */
 a++;
}                     /* end of main                */
```

Input(s) int

Output(s) double

Action int → double

Description This function converts a signed 32-bit integer to a double-precision floating-point format.

Errors None.

Example

This example of C code generates a call to _FP_ITOD:

```
main()
{
 double  a ;          /* declare one double */
 integer i ;          /* precision floating
                      point number and one
                      integer  */
 a = i ;
}                     /* end of main          */
```

Input(s) int

Output(s) float

Action int → float

Description This function converts a signed 32-bit integer to a double-precision floating-point format.

Errors None.

Example

This example generates a call to _FP_ITOF:

```
main()
{
 float   a ;          /* declare one single */
 integer i ;          /* precision floating
                      point number and one
                      integer  */

 a = i ;
}                     /* end of main          */
```

Input(s) double1, double2

Output(s) double

Action double2 - double1 → double

Description This function takes two double-precision floating-point numbers and forms their difference.

Errors None.

There are no errors generated in _FP_MINUS because the function merely negates the value of double1, then calls _FP_ADD. Errors are detected as in _FP_ADD.

Example

This example generates a call to _FP_MINUS:

```
main()
{
 double a,b,c ;        /* declare three double
                          precision floating
                          point numbers          */
 c = a - b ;
}                      /* end of main             */
```

Input(s) double1, double2

Output(s) double

Action double1 * double2 → double

Description This function takes two double-precision floating-point numbers and forms their product.

Errors Overflow (error code # = 6)
Underflow (error code # = 13)
Infinity * zero (error code # 11)

Example

This example generates a call to _FP_MUL:

```
main()
{
 double a,b,c ;        /* declare three double
                          precision floating
                          point numbers          */
 c = a * b ;
}                      /* end of main             */
```

Input(s) double

Output(s) double

Action -(double) → double

Description This function negates the value of a double-precision floating-point number.

Errors None.

Example

This example generates a call to _FP_NEG.

```
main()
{
 double a,b ;          /* declare two double
                          precision floating
                          point numbers          */
 a = - b ;
}                      /* end of main              */
```

Input(s) unsigned int

Output(s) double

Action unsigned int → double

Description This function converts an unsigned 32-bit integer to double-precision floating-point format.

Errors None.

Example

This example of C code generates a call to _FP_UTOD:

```
main()
{
 double  a ;          /* declare one double */
 unsigned j ;         /* precision floating
                         point number and one
                         integer  */
 a = j ;
}                     /* end of main          */
```

Input(s) unsigned int

Output(s) float

Action unsigned int → float

Description This function converts an unsigned 32-bit integer to single-precision floating-point format.

Errors None.

Example

This example generates a call to _FP_UTOF:

```
main()
{
 float    a ;          /* declare one single */
 unsigned j ;          /* precision floating
                          point number and one
                          integer  */

 a = j ;
}                      /* end of main           */
```

E. Interfacing Assembly Language with C

This appendix is an example of the operation of the TMS34010 C calling sequence. It serves as a guide for users who wish to interface assembly language routines with routines written in GSP C. The assembly code shown herein was generated by passing the set of C routines below through the TMS34010 C compiler, which produced TMS34010 assembly language output. You are encouraged to modify the C routines, recompile them, and study the resultant assembly language output to see the effects of the modification.

```
/* global definition */
int var3;
long var5;

main ()
{
 char var1;
 int  var2;
 long var4;
 struct
   {
    int struct_element1;
    int struct_element2;
    char struct_element3 [12];
   } struct1;

 /* call a function with no parameters */
 var1 = function1 ();

 /* call a function with parameters */
 function2 (var1, var2, var3, var4, var5);

 /* call a function with a structure as a parameter */
 function3 (var1, struct1, var2);

}

function1 ()

/* no paramters */

{
 return (10);
}
```

```
function2 (parm1, parm2, parm3, parm4, parm5)

char parm1;
int  parm2, parm3;
long parm4, parm5;

{
 /* local variables */
 int local variable;

 local_variable = parm1;
 local_variable = parm2;
 local_variable = parm3;
 local_variable = parm4;
 local_variable = parm5;

 return;
}

function3 (parm1, struct_parm, parm2)

char parm1;
struct
   {
    int struct_element1;
    int struct_element2;
    char struct_element3 [12];
   } struct_parm;
int  parm2;

{
 /* local variables */
 int local variable;
 char local variable2;

 local_variable = parm1;
 local_variable = struct_parm.struct_element1;
 local_variable = struct_parm.struct_element3 [10];
 local_variable2 = parm2;

 return;
}
```

The following assembly language code was produced by the TMS34010 C compiler from the C routines on the previous pages. The comments were added for your convenience.

```
*******************************************************
*       GSP C COMPILER,   Version 1.04, 86.200        *
*******************************************************
FP      .set    A13
STK     .set    A14
        .file   "foo.c"
        .globl  _var3
        .bss    _var3,32,16
        .globl  _var5
        .bss    _var5,32,16
        .globl  _main
*******************************************************
* FUNCTION DEF : _main
*******************************************************
 main:
        MMTM    SP,A7,FP
        MOVE    STK,FP
        ADDI    240,STK         ;allocate space for local variables

; Since the first subroutine call has no parameters there is no preparation
        CALLA   _function1      ;function returns int in A8
        MOVE    A8,A7           ;return value in A8
        MOVB    A7,*FP          ;make assignment to var1 (*FP(0))

; Prepare for second subroutine call
        MOVE    STK,-*SP,1      ;Push the program stack pointer onto the
                                ;   hardware stack. This allows the called
                                ;   routine to be the one to clean up the
                                ;   parameters and its local data area. This
                                ;   value is simply loaded from the hardware
                                ;   stack back into the system stack.
        MOVE    @_var5,*STK+,1  ;Push global variable var5 onto stack
        MOVE    *FP(48),*STK+,1 ;Push local variable var4 onto stack
        MOVE    @_var3,*STK+,1  ;Push global variable var3 onto stack
        MOVE    *FP(16),*STK+,1 ;Push local variable var2 onto stack
        MOVB    *FP,A7          ;Push local variable var1 (byte) onto stack
        MOVE    A7,*STK+,1
        CALLA    function2      ;Make call

; Prepare for third subroutine call
        MOVE    STK,-*SP,1
        MOVE    *FP(16),*STK+,1 ;Push local variable var2 onto stack
```

```
;       Push local variable struct1 (struct) onto stack
        MOVE    FP,A7
        ADDI    80,A7
        MOVK    5,A8
LL3:                            ;Loop to copy struct1 onto stack
        MOVE    *A7+,*A14+,1
        DSJS    A8,LL3

        MOVB    *FP,A7          ;Push local variable var1 (byte) onto stack
        MOVE    A7,*STK+,1
        CALLA   _function3      ;Make call

EPI0_1:
        ADDI    -240,STK        ;Readjust frame pointer to free locals
        MMFM    SP,A7,FP        ;Restore all registers that were used
        RETS    0               ;Pop off return address only


        .globl  _function1
*******************************************************
* FUNCTION DEF : _function1
*******************************************************
 function1:
        MOVK    10,A8           ;return (10)
EPI0_2:
        RETS    0               ;Pop off return address only

        .globl  _function2
*******************************************************
* FUNCTION DEF : _function2
*******************************************************
_function2:
        MMTM    SP,A7,FP            ;Save all registers to be used
        MOVE    STK,FP              ;Create new local frame
        ADDK    32,STK              ;Only one 32 bit variable
        MOVB    *FP(-32),A7         ;Extract parm1 (8 bit char)
        MOVE    A7,*FP(0),1         ;local variable = parm1
        MOVE    *FP(-64),*FP(0),1   ;local variable = parm2
        MOVE    *FP(-96),*FP(0),1   ;local variable = parm3
        MOVE    *FP(-128),A7,1      ;local variable = parm4
        MOVE    A7,*FP,1
        MOVE    *FP(-160),A7,1      ;local variable = parm5
        MOVE    A7,*FP,1
```

```
EPI0_3:
        MOVE    *SP(96),STK,1       ;Functions that have parameters must clean
                                    ;   up their own stack. This is done by
                                    ;   placing the old system stack pointer
                                    ;   back into into STK. The 96 comes from
                                    ;   64 bits (A7 and FP) and 32 bits for
                                    ;   the return PC.
        MMFM    SP,A7,FP            ;Restore all registers that were used
        RETS    2                   ;Pop off the return PC and the two words
                                    ;   of the old system stack pointer.


        .globl  _function3
*******************************************************
* FUNCTION DEF : _function3
*******************************************************
 function3:
        MMTM    SP,A7,FP              ;Save all registers to be used
        MOVE    STK,FP                ;Create new local frame
        ADDI    48,STK                ;One 32 bit variable and one char, chars
                                      ;   are stored in 16 bits for efficiency

        MOVB    *FP(-32),A7           ;Extract parm1 (8 bit char)
        MOVE    A7,*FP,1

        MOVE    *FP(-192),*FP(0),1    ;local variable =
                                      ;   struct_parm.struct_element1
        MOVB    *FP(-48),A7           ;local variable =
        MOVE    A7,*FP,1              ;   struct_parm.struct_element3[10]

        MOVE    *FP(-224),A7,1        ;local variable2 =
        MOVB    A7,*FP(32)            ;   parm2
EPI0_4:
        MOVE    *SP(96),STK,1
        MMFM    SP,A7,FP
        RETS    2
        .end
```

Index

A

B

C

D

E

F

G

I

K

Revision *
December 1986
Printed in U.S.A.

TEXAS INSTRUMENTS

SPVU005